Copy No. 9

DEFENCE R&D DÉFENSE

C++ CLASSES FOR SPARSE MATRICES AND ITERATIVE LINEAR SOLVERS

David Hally

DEFENCE RESEARCH ESTABLISHMENT ATLANTIC

Technical Memorandum

DREA TM 98/240

October 1998

National Defence Défense nationale

Canada

102-02885/

National Defence
Research and Development Branch

Défense nationale
Bureau de recherche et développement

C++ CLASSES FOR SPARSE MATRICES AND ITERATIVE LINEAR SYSTEM SOLVERS

David Hally

October 1998

Approved by: J.L. Kennedy:
Head / Warship Signatures and Safety Section

TECHNICAL MEMORANDUM 98/240

Defence Research Establishment Atlantic

Centre de Recherches pour la Défense Atlantique

Canada

Abstract

C++ classes for representing sparse matrices are described. The matrices use one of four profile storage schemes: compressed row storage, skyline storage, sparse skyline storage (similar to skyline storage but with compressed arrays above and below the diagonal), and S2dBlock storage (a special purpose storage scheme for matrices arising from partial differential equations on two-dimensional structured blocks).

C++ classes are also provided for iterative solution methods for linear systems of equations. The Conjugate Gradient, Conjugate Gradient Squared, Quasi-Minimum Residual, and Generalized Minimum Residual (GMRES) methods have been implemented. Element re-ordering and preconditioning using the diagonal (Jacobi) or incomplete LU factorization of order zero (ILU0) methods are also provided.

Résumé

Description des classes d'objets C++ pour représenter des matrices d'analyse. Les matrices utilisent l'un des quatre systèmes de stockage des profils: par rangées comprimées, par horizon d'analyse (similaire à l'horizon, mais avec des tableaux comprimés au-dessus et au dessous de la diagonale) et par bloc S2d (système de stockage à but spécial pour des matrices provenant d'équations différentielles partielles sur blocs structurés bidimensionels).

Les classes d'objects C++ sont également fournies pour des méthodes de solution itérative pour les systèmes linéaires d'équations. Les méthodes du gradient conjugé, du gradient congugé au carré, de la valeur résiduelle quasi-minimum (QMR) et de la valeur résiduelle minimum généralisée (GMRES) ont été mises en oeuvre. La remise en ordre et le préconditionnement des éléments en utilisant les méthodes de la diagonale (Jacobi) ou de la factorisation LU incomplète de l'ordre zéro (ILU0) sont également donnés.

DREA Technical Memorandum TM/98/240

C++ Classes for Sparse Matrices and Iterative Linear System Solvers

by

David Hally

Executive Summary

Background

The computer program TRANSOM is intended to solve a number of ship-related flow problems of interest to DREA: for example, flows around submarines, vortex generation from propellers and control surfaces, bilge vortex generation, and prediction of roll damping coefficients. In particular it will enhance our ability to predict the flow around a ship hull and into the propellers. Knowledge of this flow is crucial for the design of efficient quiet propellers.

TRANSOM allows the use of different solution methods on different regions of the flow. Each of these methods uses a large linear system of equations which must be evaluated and solved repeatedly in order to determine the flow. In early versions of the program each solver used a separate representation of the linear system. The current memorandum describes a single set of C++ classes which can be used to store and solve the linear systems for any of the solutions methods.

Principal Results

C++ classes have been written for representing sparse matrices and iterative solvers for linear systems which use them. The matrices use one of four profile storage schemes: compressed row storage, skyline storage, sparse skyline storage (similar to skyline storage but with compressed arrays above and below the diagonal), and S2dBlock storage (a special purpose storage scheme for matrices arising from partial differential equations on two-dimensional structured blocks).

C++ classes are also provided for iterative solution methods for linear systems of equations. The Conjugate Gradient, Conjugate Gradient Squared, Quasi-Minimum Residual, and Generalized Minimum Residual (GMRES) methods have been implemented. Element re-ordering and preconditioning using the diagonal (Jacobi) or incomplete LU factorization of order zero (ILU0) methods are also provided.

While developed for use in TRANSOM, these classes will find much wider application; they can be used in any C++ program which uses large sparse matrices or large linear systems of equations.

Significance of Results

The matrix and linear system classes greatly improve the storage and solution of linear systems in TRANSOM. The new classes are much more general and more efficient than earlier implementations. Programming time is reduced by having a single set of classes for all solvers. Moreover, the functionality of TRANSOM has been enhanced since the new matrix classes allow the use of a single global matrix which can be shared by solvers of different types. The result is a program which is easier to maintain and more adaptable, therefore more likely to meet the requirement of solving the difficult flows encountered in naval applications.

Contents

Abstract ii

Executive Summary iii

Table of Contents v

List of Figures vii

1 Introduction 1

2 Theory 2

2.1 Sparse Matrices 2

2.1.1 The CRS Storage Scheme 2

2.1.2 The Skyline Storage Scheme 3

2.1.3 The Sparse Skyline Storage Scheme 5

2.1.4 The S2dBlock Storage Scheme 6

2.2 Linear System Solvers 8

2.2.1 Preconditioning 9

2.2.2 Element Reordering 11

3 Using the Sparse Matrix Classes 12

3.1 Vectors 12

3.2 Sparse Matrices 15

3.2.1 The Sparse Matrix Base Class 17

3.2.2 Access to the Matrix Elements 18

3.2.3 Memory Management 19

3.2.4 Iterators over the Non-zero Elements 19

3.2.5 Arithmetic Operators 21

3.2.6 Specializations of `SpMtx` 22

3.3 Preconditioners 25

3.3.1 Class `PreDiagSpMtx` 26

3.3.2 Class `PreILUOSpMtx`: 26

3.4 Linear System Solvers 27

3.4.1 The SpLinSys Class 30
3.4.2 Element Reordering 32
3.4.3 Node Mapping 35
3.4.4 The NodeMapSpLinSys Class 37
3.4.5 The S2dSpLinSys Class 38

4 Implementation of the Sparse Matrix Classes 40
4.1 Shared Representations 40
4.1.1 The ArrayRep Class 43
4.2 Vectors 43
4.3 Sparse Matrices 45
4.3.1 The Classes SpMtxProf and SpMtxProfRep 45
4.3.2 The SpMtx and SpMtxRep Class 48
4.3.3 Sparse Matrix Iterators 50
4.3.4 The CRS Matrix 52
4.3.5 The Skyline Matrix 56
4.3.6 The Sparse Skyline Matrix 56
4.3.7 The S2dBlock Matrix 57
4.4 Preconditioning Matrices 57
4.4.1 ILU0 Preconditioning Matrices 58
4.5 Linear System Solvers 58
4.5.1 Preconditioning 64
4.5.2 Linear System Solvers with Choice of Method 67
4.5.3 The Classes SpLinSys, NodeMapSpLinSys, and S2dSpLinSys 68

5 Concluding Remarks 71

References 72

List of Figures

1 A 5 × 3 O-block: (i, j) indexing. 6
2 A 5 × 3 O-block: node indexing. 6
3 Public interface for the class `Vector`. 13
4 Public interface for the class `SpMtx`. 16
5 Public interface for the sparse matrix iterator classes. 20
6 Arithmetic functions for matrix elements. 21
7 Public interface for the class `CRSMtx`. 22
8 Public interface for the class `S2dBlockMtx`. 22
9 Public interface for the class `SpSkyMtx`. 23
10 Public interface for the class `SkyMtx`. 23
11 Public interface for the preconditioner classes. 25
12 Public interface for the linear system solver classes. 28
13 Public interface for the class `SpLinSys`. 31
14 Public interface for the class `Permutation`. 33
15 Public interface for the classes `PermVector` and `PermSpMtx`. 34
16 Public interface for the node mapping classes. 36
17 Public interface for the class `NodeMapSpLinSys`. 37
18 Public interface for the class `S2dSpLinSys`. 38
19 Class diagram for shared representations. 40
20 Interface for the class `RefCount`. 41
21 Interface for the class `Handle`. 42
22 Interface for the class `ArrayRep`. 43
23 Class diagram for vectors. 44
24 Class diagram for sparse matrices. 46
25 Interface for the class `SpMtxProfRep`. 47
26 Interface for the class `SpMtxProf`. 48
27 Interface for the class `SpMtxRep`. 49
28 Interface for the class `SpMtx`. 51
29 Interface for the class `SpMtxProfIter`. 52
30 Interface and implementation of the classes `SpMtxIter` and `ConstSpMtxIter`. 53
31 Class diagram for the CRS matrix. 54

32 Interface for the class CRSMtxRep. 55
33 Class diagram for linear system solvers without preconditioning. 59
34 Interfaces for the classes LinSysSweep and LinSysSolver. 60
35 Interfaces for residual linear system solver classes. 62
36 Interface for the class NResSolver. 63
37 Interface for the class PreMtx. 64
38 Class diagram for PreLinSysSolver. 66
39 Interface for the class PreLinSysSolver. 67
40 Interfaces for the classes NoTransLinSysSolver and TransLinSysSolver. 69
41 Full interface for the class SpLinSys. 70

1 Introduction

This memorandum describes C++ classes for representing sparse matrices and for solving linear systems which use them. These classes were developed for use by the flow solver TRANSOM but will find much wider application; they can be used in any C++ program which uses large sparse matrices or large linear systems of equations.

TRANSOM is a Reynolds-averaged Navier-Stokes solver developed at DREA for predicting the flow around ships, submarines and propellers[1]. TRANSOM allows different solution methods to be applied to different regions of the flow. In particular, it can use a pseudo-compressibility method (SPC) on structured blocks[2] and a finite element method (FE) on unstructured blocks[3]. In the early versions of TRANSOM these two solvers used different representations for the matrices in their linear systems: the SPC method used a restricted version of compressed diagonal storage, while the FE method used a skyline matrix. In addition to the extra effort required to maintain two separate matrix representations, these classes had several disadvantages.

1. The SPC matrix, having originated in Fortran code, was not well encapsulated, its data structures being used directly by several important functions.
2. The profile of the SPC matrix was limited to five diagonals, thus allowing only five non-zero elements on each row. This precluded implementation of boundary conditions for which the boundary node depends on more than four of its neighbours.
3. The SPC matrix was limited to use on a single block precluding the use of a single global matrix for a multi-block grid.
4. The FE matrix skyline structure was very wasteful of memory.
5. The SPC and FE solvers could not be combined into a single solver sharing a global matrix.

The classes described here overcome these problems by providing sparse matrix classes which have a consistent interface but which allow different storage schemes. Preliminary work on the implementation of sparse matrices for the FE solver is described in Reference 4, Section 7.1.6. The classes described here are a much more general implementation.

This memorandum has three principal sections. Section 2 provides an overview of the principles of the sparse matrix storage schemes and of the linear system solvers. Section 3 provides the information a programmer needs to use the classes in a C++ program. Section 4 provides details of the way in which the classes have been implemented.

2 Theory

2.1 Sparse Matrices

In a sparse matrix almost all the elements are zero. Substantial reductions in memory usage and CPU time can be obtained if only the non-zero elements of the array are stored and used in computations.

The profile of a sparse matrix is a list of its non-zero elements. To be more precise, it is a list of its elements which *can be* non-zero, since it is always possible that one of the elements in the profile is assigned to zero.

The use of sparse matrices is most efficient if the elements in the profile are stored contiguously in memory without wasted space for the zeros. Of course, this requires an indexing scheme to determine where the value of an element in the profile is stored. Many such schemes have been proposed: for examples, see Reference 5, Chapter 4. Four different schemes have been implemented in TRANSOM and will be described in detail here:

1. Compressed row storage (CRS): This scheme is completely general; it can be used to represent any matrix profile whatsoever.

2. Skyline storage: This scheme is also completely general but may only be used for square matrices. Although it is significantly more costly in memory than the other three storage schemes, this is the only representation in TRANSOM which allows the direct non-iterative solution of linear systems via Gaussian elimination or some similar algorithm. It is therefore suitable for the solution of very poorly conditioned linear systems such as those which arise from penalty function methods in finite element flow solvers[3].

3. Sparse skyline storage: This is a more memory efficient version of the skyline storage scheme. It is completely general but can only be used for square matrices.

4. S2dBlock storage: This is a variant of compressed diagonal storage suitable for solvers on structured two-dimensional blocks. The S2dBlock scheme corresponds to the matrix representation used by the early versions of TRANSOM and its predecessor NSI2D[6] and has a severely limited profile: many elements in the matrix *must* be zero. However, for solvers on single two-dimensional structured blocks, it provides a matrix which is very efficient in both memory requirements and calculation time.

2.1.1 The CRS Storage Scheme

The CRS storage scheme stores the elements of the profile row by row. Thus, the array

$$\begin{pmatrix} 1 & 0 & 8 & 0 & 0 & 11 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & -5 & 0 & 7 & 0 & 0 \\ 0 & 0 & 0 & -2 & 9 & 10 \\ 0 & 3 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.1}$$

is stored in an array values as

$$(1, 8, 11, 4, -5, 7, -2, 9, 10, 3) \tag{2.2}$$

Two arrays of integer pointers are required to determine the element of the matrix which corresponds to each of these values. The array cols contains the number of the column for each element in the profile. In the example above, cols would have the values (the first column is column zero):

$$(0, 2, 5, 0, 1, 3, 3, 4, 5, 1) \tag{2.3}$$

The array row_start is used to specify which elements of values are on each row of the matrix. The non-zero elements on row i are stored in values between row_start[i] and row_start[i+1]-1. Thus, except for the case when row i has no non-zero elements, row_start[i] is the location in values of the first non-zero element on row i. For an empty row, row_start[i] is set to row_start[i-1]. So that row_start[i+1] is defined for the last row of the matrix, the length of row_start is set to be one greater than the number of rows and row_start[i+1] is set to the number of non-zero elements in the matrix. In the example above, row_start would have the values:

$$(0, 3, 4, 6, 9, 10) \tag{2.4}$$

The value of element (m, n) may be determined using the following algorithm:

```
for (k = row_start[m]; k < row_start[m+1]; k++)
  if      (cols[k] == n) return values[k];
  else if (cols[k] >  n) return 0;
return 0;
```

If there were a large number of elements on each row, a faster method of searching for column n might be worthwhile. Here it is assumed that the simple linear search is just as fast. This is the case for the matrices used in TRANSOM, typically having fewer than ten elements per row.

2.1.2 The Skyline Storage Scheme

The skyline storage scheme can be used only for square matrices. In this scheme the elements on the diagonal, above the diagonal, and below the diagonal are stored in three separate arrays.

The elements on the diagonal always belong to the matrix profile: i.e. the value of a diagonal element is always stored, even if it is known to be zero.

The elements below the diagonal are stored in row order similar to the CRS scheme except that, on each row, the values of all elements between the first non-zero element and the diagonal are stored. Above the diagonal the elements are stored in column order. Again, values are stored for all elements between the first non-zero element on a column and the diagonal element.

In the following matrix the boundary surrounds the elements whose values are stored above and below the diagonal.

$$\begin{pmatrix} 1 & 9 & 8 & 10 & 0 & 0 \\ 4 & 2 & 0 & 0 & 0 & 5 \\ 3 & 5 & 6 & 7 & 8 & 0 \\ 0 & 4 & 1 & 2 & 9 & 0 \\ 3 & 3 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 2 \end{pmatrix} \tag{2.5}$$

The values of the elements on, below, and above the diagonal are stored in the three arrays `diag_vals`, `lower_vals`, and `upper_vals`, respectively. For the example above one has:

$$\texttt{diag\_vals} = (1,2,6,2,1,2) \tag{2.6}$$

$$\texttt{lower\_vals} = (4,3,5,4,1,3,3,0,0,2,0,0) \tag{2.7}$$

$$\texttt{upper\_vals} = (9,8,0,10,0,7,8,9,5,0,0,0) \tag{2.8}$$

For each of the upper and lower profiles, a pointer array is also needed to indicate the first non-zero element on each column or row. For the lower profile the pointer array is denoted `lrow_start` and for the upper profile it is `ucol_start`. Each has length one greater than the size of the matrix. The elements on row `i` below the diagonal are stored in `lower_vals` between `lrow_start[i]` and `lrow_start[i+1]-1`. Similarly, elements in column `j` above the diagonal are stored in `upper_vals` between `ucol_start[j]` and `ucol_start[j+1]-1`. In the example above, `lrow_start` and `ucol_start` would have the values:

$$\texttt{lrow\_start} = (0,0,1,3,5,9,12); \qquad \texttt{ucol\_start} = (0,0,1,3,6,8,12) \tag{2.9}$$

The value of element (m,n) may be determined using the following algorithm:

```
if (m == n)
  return diag_vals[m];
else if (m > n) {
  k = lrow_start[m+1] - m + n;
  if (k < lrow_start[m]) return 0;
  else                   return lower_vals[k];
}
else {
  k = ucol_start[n+1] - n + m;
  if (k < ucol_start[n]) return 0;
  else                   return upper_vals[k];
}
```

The profile of a sparse matrix is symmetric if (n,m) is in the profile whenever (m,n) is. When the profile is symmetric, the arrays `lrow_start` and `ucol_start` are identical. This can be used to reduce the memory overhead for matrices with symmetric profiles.

Similarly, when the matrix is symmetric, the arrays `lower_vals` and `upper_vals` are identical and the storage requirements can be significantly reduced.

2.1.3 The Sparse Skyline Storage Scheme

The sparse skyline storage scheme is similar to the skyline storage scheme in that the elements on the diagonal are stored separately, the elements below the diagonal are stored in row order, and the elements above the diagonal are stored in column order. The difference is that the elements above and below the diagonal are stored in compressed format (similar to the CRS scheme) so that values are stored only for the non-zero elements. This typically requires much less memory, but precludes the use of a direct non-iterative solution of a linear system using the matrix.

As in the skyline storage scheme, the values of the elements on, below, and above the diagonal are stored in the three arrays `diag_vals`, `lower_vals`, and `upper_vals`. For the example in Equation (2.5) one has:

$$\texttt{diag\_vals} = (1,2,6,2,1,2) \tag{2.10}$$

$$\texttt{lower\_vals} = (4,3,5,4,1,3,3,2) \tag{2.11}$$

$$\texttt{upper\_vals} = (9,8,10,7,8,9,5) \tag{2.12}$$

The arrays `lrow_start` and `ucol_start` give the locations of the elements in each row and column. In the example:

$$\texttt{lrow\_start} = (0,0,1,3,5,7,8); \qquad \texttt{ucol\_start} = (0,0,1,2,4,6,7) \tag{2.13}$$

Unlike the skyline storage scheme, two extra arrays are needed to give the row numbers of the elements above the diagonal, and the column numbers of the elements below the diagonal. These arrays are denoted `rows` and `cols`, respectively. In the example their values are:

$$\texttt{rows} = (0,0,0,2,2,3,1); \qquad \texttt{cols} = (0,0,1,1,2,0,1,2) \tag{2.14}$$

The value of element (m,n) may be determined using the following algorithm:

```
if (m == n)
  return diag_vals[m];
else if (m > n) {
  for (k = lrow_start[m]; k < lrow_start[m+1]; k++)
    if      (cols[k] == n) return lower_vals[k];
    else if (cols[k] >  n) return 0;
  return 0;
}
else if (m < n) {
  for (k = ucol_start[n]; k < ucol_start[n+1]; k++)
    if      (rows[k] == m) return upper_vals[k];
    else if (rows[k] >  m) return 0;
  return 0;
}
```

Like the skyline storage scheme, the sparse skyline storage scheme allows reduction in memory requirements for symmetric matrices and matrices with symmetric profiles. When the

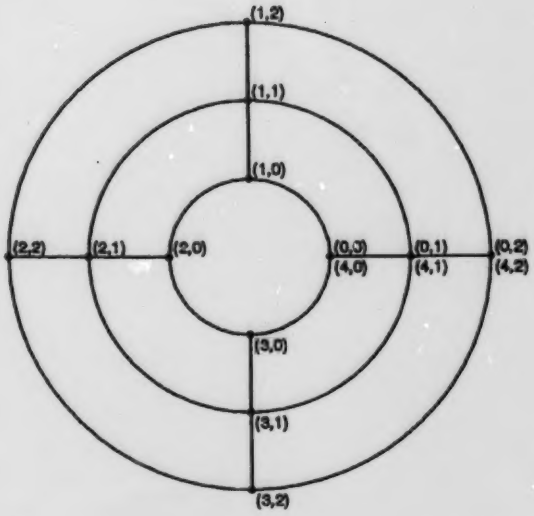

Figure 1: *A 5 × 3 O-block: (i,j) indexing.*

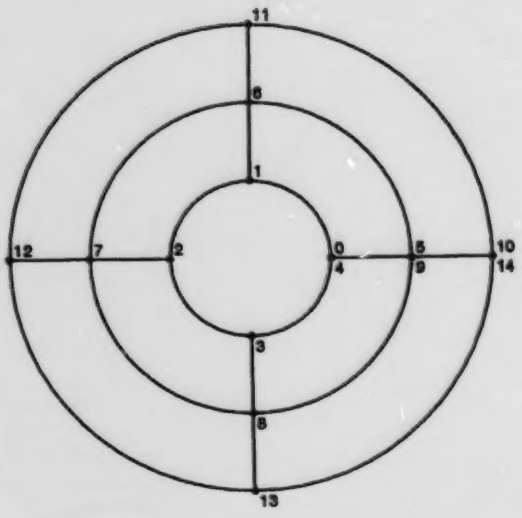

Figure 2: *A 5 × 3 O-block: node indexing.*

profile is symmetric, the arrays `lrow_start` and `cols` are identical to the arrays `ucol_start` and `rows`.

Similarly, when the matrix is symmetric, the arrays `lower_vals` and `upper_vals` are identical and the storage requirements can be significantly reduced.

2.1.4 The S2dBlock Storage Scheme

The S2dBlock storage scheme is a special purpose scheme suitable for the solution of differential equations on an $M \times N$ structured block. In such a block, the nodes can be assigned indices (i,j) where i ranges between 0 and $M-1$ and j ranges between 0 and $N-1$. The node numbers, n, run from 0 to $MN-1$ and are related to i and j by:

$$n = i + Mj \tag{2.15}$$

The S2dBlock matrix is suitable for linear systems in which variables at each node of the block depend only on their four immediate neighbours: nodes $(i-1,j)$, $(i+1,j)$, $(i,j-1)$, and $(i,j+1)$. Thus, each row of the matrix has (at most) five non-zero elements. Using Equation (2.15), we find that the non-zero elements of row n of the array are $(n,n-M)$, $(n,n-1)$, (n,n), $(n,n+1)$, and $(n,n+M)$. Therefore, the matrix has five principal diagonal bands. The matrix elements along each of these bands are stored in five arrays: `a1band`, `b1band`, `c1band`, `d1band`, and `e1band`, respectively.

For points lying on the boundaries, one or more of the neighbouring nodes may not exist. Moreover, for blocks with re-entrant boundaries, one of the neighbouring nodes may not lie in one of the principal bands. For example, consider the 5 × 3 block with O-topology in Figures 1 and 2: The nodes with $i = 4$ are duplicates of those with $i = 0$. The neighbours of node 5

are nodes 0, 6, 8, and 10. The three nodes 0, 6, and 10 lie on the bands a1band, b1band, and e1band, but node 8 does not lie on d1band. Moreover, node 11 has only three neighbours: 6, 9, and 10.

The S2dBlock matrix use four arrays, one for each boundary, to store the numbers of the neighbouring nodes across the boundary. If there is no neighbouring node across the boundary the node number -1 is used as a flag. The arrays are denoted ilo_bptr, ihi_bptr, jlo_bptr, and jhi_bptr. For the O-block example these arrays have the values:

$$\texttt{ilo\_bptr} = (3, 8, 13) \tag{2.16}$$
$$\texttt{ihi\_bptr} = (1, 6, 11) \tag{2.17}$$
$$\texttt{jlo\_bptr} = (-1, -1, -1, -1, -1) \tag{2.18}$$
$$\texttt{jhi\_bptr} = (-1, -1, -1, -1, -1) \tag{2.19}$$

When an element of ilo_bptr[j] is not -1, then the matrix element (M*j,ilo_bptr[j]) can be non-zero. The value of this element is stored in b1band[M*j]. Similarly, when ihi_bptr, jlo_bptr, and jhi_bptr have elements which are not not -1, the corresponding matrix element values are stored in d1band, a1band, and e1band, respectively.

The following matrix conforms to the structure required by an S2dBlock matrix for the block in Figures 1 and 2. To highlight the structure of the matrix, zeros have been replaced by dots.

$$\left(\begin{array}{ccccc|ccccc|ccccc}
8 & 2 & . & 7 & . & 1 & . & . & . & . & . & . & . & . & . \\
1 & 3 & 1 & . & . & . & 2 & . & . & . & . & . & . & . & . \\
. & 3 & 9 & 3 & . & . & . & 1 & . & . & . & . & . & . & . \\
. & . & 4 & 5 & 3 & . & . & . & 3 & . & . & . & . & . & . \\
. & 6 & . & 1 & 5 & . & . & . & . & 4 & . & . & . & . & . \\
\hline
1 & . & . & . & . & 6 & 1 & . & 7 & . & 4 & . & . & . & . \\
. & 1 & . & . & . & 2 & 7 & 1 & . & . & . & 2 & . & . & . \\
. & . & 2 & . & . & . & 3 & 8 & 3 & . & . & . & 3 & . & . \\
. & . & . & 2 & . & . & . & 2 & 3 & 1 & . & . & . & 1 & . \\
. & . & . & . & 3 & . & 6 & . & 1 & 7 & . & . & . & . & 1 \\
\hline
. & . & . & . & . & 3 & . & . & . & . & 8 & 2 & . & 7 & . \\
. & . & . & . & . & . & 1 & . & . & . & 1 & 6 & 3 & . & . \\
. & . & . & . & . & . & . & 2 & . & . & . & 3 & 8 & 4 & . \\
. & . & . & . & . & . & . & . & 2 & . & . & . & 2 & 8 & 3 \\
. & . & . & . & . & . & . & . & . & 1 & . & 6 & . & 2 & 9
\end{array}\right) \tag{2.20}$$

The values stored in the band arrays are as follows:

$$\texttt{a1band} = (*, *, *, *, *, 1, 1, 2, 2, 3, 3, 1, 2, 2, 1) \tag{2.21}$$
$$\texttt{b1band} = (7, 1, 3, 4, 1, 7, 2, 3, 2, 1, 7, 1, 3, 2, 2) \tag{2.22}$$
$$\texttt{c1band} = (8, 3, 9, 5, 5, 6, 7, 8, 3, 7, 8, 6, 8, 8, 9) \tag{2.23}$$
$$\texttt{d1band} = (2, 1, 3, 3, 6, 1, 1, 3, 1, 6, 2, 3, 4, 3, 6) \tag{2.24}$$
$$\texttt{e1band} = (1, 2, 1, 3, 4, 4, 2, 3, 1, 1, *, *, *, *, *) \tag{2.25}$$

The asterisks indicate that an element of the array is never used. The values in bold face type are those which lie outside the principal bands.

The value of matrix element (m, n) may be determined using the following algorithm:

```
if (m == n) return c1band[m];

// Find (i,j) corresponding to node m
i = m%M; j = m/M;

// Check for elements in the default profile
if (n == m-1 && i != 0  ) return b1band[m];
if (n == m+1 && i != M-1) return d1band[m];
if (n == m-M && j != 0  ) return a1band[m];
if (n == m+M && j != N-1) return e1band[m];

// Check for points on the boundaries
if (j == 0) {
  if (i == 0   && ilo_bptr[j] == n) return b1band[m];
  if (i == M-1 && ihi_bptr[j] == n) return d1band[m];
  if (jlo_bptr[i] == n) return a1band[m];
  return 0;
}
if (j == N-1) {
  if (i == 0   && ilo_bptr[j] == n) return b1band[m];
  if (i == M-1 && ihi_bptr[j] == n) return d1band[m];
  if (jhi_bptr[i] == n) return e1band[m];
  return 0;
}
if (i == 0) {
  if (ilo_bptr[j] == n) return b1band[m];
  return 0;
}
if (i == M-1) {
  if (ihi_bptr[j] == n) return d1band[m];
  return 0;
}
return 0;
```

2.2 Linear System Solvers

The primary use for sparse matrices is to represent the matrix in a large linear system.

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{2.26}$$

Given matrix **A** and vector **b**, the vector **x** is to be determined. Reference 5 discusses several iterative solution methods which have the property that they use the matrix **A** only via a matrix-vector product or, sometimes, via the product of the transpose of **A** and a vector. Of the

methods discussed there, Conjugate Gradient (CG), Conjugate Gradient Squared (CGS), Quasi-Minimum Residual (QMR) with no look-ahead, and Generalized Minimum Residual (GMRES) have been implemented in TRANSOM.

In addition to the methods described in Reference 5, the Successive Substitution (SS) method has also been implemented. Suppose that $\mathbf{x}_n$ is the n^{th} approximation of the solution vector. We look for $\Delta\mathbf{x}$ such that $\mathbf{x}_n + \Delta\mathbf{x}$ is the solution. Using Equation (2.26) we find the following expression for $\Delta\mathbf{x}$:

$$\Delta\mathbf{x} = \mathbf{A}^{-1}(\mathbf{b} - \mathbf{A}\cdot\mathbf{x}_n) \tag{2.27}$$

In the successive substitution method it is supposed that the identity matrix is an adequate approximation of $\mathbf{A}^{-1}$. Then a new approximation to the solution vector is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + (\mathbf{b} - \mathbf{A}\cdot\mathbf{x}_n) \tag{2.28}$$

It is clear that this method will converge only when $\mathbf{A}$ closely approximates the identity. In fact, its main use is to test the effectiveness of different preconditioning schemes (see Section 2.2.1).

An important property of each of these solution methods is that they use the matrix only via the matrix-vector product and, in the case of the QMR method, the product of a vector by the transpose of the matrix. Therefore they can be used with any matrix representation which defines these two operations.

2.2.1 Preconditioning

The convergence rate of the iterative linear system solvers depends strongly on the condition of the matrix $\mathbf{A}$. Improvements in convergence can usually be achieved by preconditioning the matrix.

When the matrix is preconditioned, the linear system is transformed to a different system with better condition but having the same solution vector. Let $\mathbf{M}_1$ and $\mathbf{M}_2$ be two invertible matrices and consider the linear system

$$\mathbf{A}'\mathbf{x}' = \mathbf{b}' \tag{2.29}$$

where

$$\mathbf{A}' = \mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}; \qquad \mathbf{x}' = \mathbf{M}_2\mathbf{x}; \qquad \mathbf{b}' = \mathbf{M}_1^{-1}\mathbf{b} \tag{2.30}$$

It has the same solution vector, $\mathbf{x}$, as $\mathbf{A}\mathbf{x} = \mathbf{b}$. Therefore, any of the iterative solution methods can be converted to an equivalent solution method as follows:

1. Calculate $\mathbf{b}'$ by solving $\mathbf{M}_1\mathbf{b}' = \mathbf{b}$.
2. Calculate $\mathbf{x}'$ by applying the iterative method replacing $\mathbf{A}$ with $\mathbf{A}'$ and $\mathbf{b}$ with $\mathbf{b}'$.
3. Calculate the solution vector by solving $\mathbf{M}_2\mathbf{x} = \mathbf{x}'$.

In order to use $\mathbf{A}'$ in one of the linear system solvers, it is only necessary that $\mathbf{A}'\mathbf{v}$ and $\mathbf{v}\mathbf{A}'$ (equivalent to $(\mathbf{A}')^T\mathbf{v}$) be defined for any vector $\mathbf{v}$. Suppose that $\mathbf{w} = \mathbf{A}'\mathbf{v}$. Then $\mathbf{w}$ can be calculated as follows:

1. Solve $\mathbf{M_2 w_1 = v}$ for $\mathbf{w_1}$.
2. Calculate $\mathbf{w_2 = A w_1}$.
3. Solve $\mathbf{M_1 w = w_2}$ for $\mathbf{w}$.

Similarly, the product $\mathbf{w = vA'}$ can be calculated by:

1. Solve $\mathbf{w_1 M_1 = v}$ for $\mathbf{w_1}$.
2. Calculate $\mathbf{w_2 = w_1 A}$.
3. Solve $\mathbf{w M_2 = w_2}$ for $\mathbf{w}$.

Therefore, to be used in any of the linear system solvers, a representation of the preconditioning matrices need only be able to solve the linear systems $\mathbf{Mx = b}$ and $\mathbf{xM = b}$.

An effective preconditioner must have the following properties:

1. $\mathbf{A'}$ should have significantly better condition than $\mathbf{A}$.
2. One must be able to calculate the solutions to the linear systems $\mathbf{M_1 x = b}$ and $\mathbf{M_2 x = b}$ efficiently.
3. If a solution method requires multiplication by $(\mathbf{A'})^T$, then efficient solutions of $\mathbf{M_1^T x = b}$ and $\mathbf{M_2^T x = b}$ must also be available.
4. $\mathbf{M_1}$ and $\mathbf{M_2}$ should be easy to construct.
5. $\mathbf{M_1}$ and $\mathbf{M_2}$ should not require too much extra memory.

If $\mathbf{M_2}$ is the identity, then the linear system is said to be left preconditioned. In this case $\mathbf{M_1}$ should approximate $\mathbf{A}$. Similarly, if $\mathbf{M_1}$ is the identity, the system is right-preconditioned and $\mathbf{M_2}$ should approximate $\mathbf{A}$.

Two types of preconditioning have been implemented in TRANSOM: diagonal (sometimes called Jacobi), and incomplete LU factorization of degree zero (ILU0). Other preconditioners are discussed in Reference 5.

Diagonal Preconditioning: The simplest preconditioner consists of the diagonal of the matrix A:

$$M_{ij} = \begin{cases} A_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{2.31}$$

The diagonal matrix can be used for either left or right preconditioning.

ILU0 Preconditioning: In the ILU0 preconditioner, the matrix elements are calculated as in an LU decomposition of **A**, except that any non-zero element outside the profile of **A** is discarded.

$$M_{ij} = \begin{cases} A_{ij} - \sum_{k<j} A_{ik} A_{kk}^{-1} A_{kj} & \text{if } (i,j) \text{ is in the profile of } \mathbf{A} \\ 0 & \text{otherwise} \end{cases} \tag{2.32}$$

Provided that the complete diagonal lies in the profile of **A**, it can be shown that **M** may be written as $\mathbf{M} = (\mathbf{L}+\mathbf{D})\mathbf{D}^{-1}(\mathbf{U}+\mathbf{D})$ where **L** is the portion of **A** below the diagonal, **U** is the portion of **A** above the diagonal, and **D** is a diagonal matrix equal to the diagonal of **M**. In this form a linear system in **M** is easy to solve (since each of $(\mathbf{L}+\mathbf{D})$, $\mathbf{D}^{-1}$, and $(\mathbf{U}+\mathbf{D})$ is easy to solve), and requires little extra memory, since only the diagonal elements need be stored.

2.2.2 Element Reordering

In this section we consider the effect of reordering the elements of a matrix on the iterative solution methods described above.

Suppose that **A** is an $N \times N$ matrix, **x** is a vector of length N, and $p(i)$ is a permutation of $[0, \ldots, N-1]$. A new matrix $\tilde{\mathbf{A}}$ and a new vector $\tilde{\mathbf{x}}$ can be defined such that

$$\tilde{A}_{ij} = A_{p(i),p(j)}; \qquad \tilde{x}_i = x_{p(i)} \tag{2.33}$$

Consider the matrix-vector product:

$$\left(\tilde{\mathbf{A}}\tilde{\mathbf{x}}\right)_i = \sum_j \tilde{\mathbf{A}}_{ij}\tilde{\mathbf{x}}_j = \sum_j \mathbf{A}_{p(i),p(j)}\mathbf{x}_{p(j)} = \sum_j \mathbf{A}_{p(i),j}\mathbf{x}_j = \widetilde{\mathbf{Ax}}_i \tag{2.34}$$

Therefore, matrix-vector multiplication is invariant under the permutation and the iterative solution methods will yield exactly the same results (within round-off error) whether the elements are permuted or not.

Nevertheless, element re-ordering can have a significant effect on the efficiency of the matrix solver in two ways. First, when the matrix **A** contains a significant number of zero elements within its profile, the number of these elements can be reduced by node re-ordering, leading to substantial efficiency gains since many multiplications by zero are avoided. This is particularly true for the skyline storage scheme. It is well known that element re-ordering can reduce the mean bandwidth of the matrix significantly, reducing storage requirements and increasing efficiency of calculations.

Second, while the matrix-vector multiply, and hence the iterative solvers, are invariant with respect to element re-ordering, the matrix preconditioners may not be. In fact, the diagonal preconditioner is invariant while the ILU0 preconditioner is not. When ILU0 is used, node re-ordering can have a significant effect on the condition of the preconditioned matrix. Unfortunately, we do not know of a general algorithm to determine when a particular node order will be effective and when it will not. This is a subject for continuing research.

3 Using the Sparse Matrix Classes

In this section we present the information that a programmer requires to make use of the sparse matrix classes. The way in which the classes have been implemented is described in Section 4.

Many design decisions were influenced by the constraints of the C++ compiler used during the development of this code: Silicon Graphics C++ Compiler SC4-C++-4.0. In particular, exception handling was not supported so that errors are usually simply flagged by an error message; severe errors result in termination of execution. Boolean values are represented simply as `int`'s. Nested templates, default template arguments, and template member functions were not supported.

3.1 Vectors

The class `Vector` represents a numerical vector: i.e. an array of values for which arithmetic operators are defined. The complete public interface for the class is shown in Figure 3. To make the interface more readable the template declarations have been omitted for each of the operators declared after the `Vector` class. This convention will be used consistently in this memorandum where there is no possibility of confusion over template arguments.

Each element of the vector is of type `VE`. If `v1` and `v2` are of class `VE`, and `d` is a `double`, then the following operations must be defined:

1. `v1 == v2` (returns `int`).
2. `v1*d` (returns a `VE`).
3. `-v1` (returns a `VE`).
4. `zero(v1)` (set to zero).

The `Vector` class has been designed so that scalars, vectors, and matrices can all be used as elements of a vector. For this reason a generic version of the dot product of two vectors is not defined since it is not clear which operator should be applied to the elements of the vector: `operator*` is appropriate if `VE` is a scalar (a `float` or a `double`); dot product is appropriate if `VE` is a sub-vector; it is not clear what is appropriate if `VE` is a matrix. Nor is it clear what the return value should be. Similarly, the magnitude of a vector is not defined since it is not clear what should be used for the magnitude of each vector element. However, the following functions have been defined for specific instances of `VE`.

```
float  dot(const Vector<float>  &v1, const Vector<float>  &v2);
double dot(const Vector<double> &v1, const Vector<double> &v2);
double dot(const Vector<Vec2>   &v1, const Vector<Vec2>   &v2);
double dot(const Vector<Vec3>   &v1, const Vector<Vec3>   &v2);

float  abs(const Vector<float>  &v);
double abs(const Vector<double> &v);
```

```
template<class VE>
class Vector
{
public:
  Vector(unsigned n);
  Vector(const Vector<VE>&);

  Vector<VE>& operator=(const Vector<VE>&);

  unsigned len() const;

  VE& operator[](unsigned i);
  const VE& operator[](unsigned i) const;

  void discard_values();

  void zero();
  Vector<VE>& negate();
  Vector<VE>& operator+=(const Vector<VE>&);
  Vector<VE>& operator-=(const Vector<VE>&);
  Vector<VE>& operator*=(double);
  Vector<VE>& operator/=(double);

  friend int operator==(const Vector<VE>&,const Vector<VE>&);
  friend int operator!=(const Vector<VE>&,const Vector<VE>&);
};

Vector<VE> operator-(const Vector<VE>&);

Vector<VE> operator+(const Vector<VE>&,const Vector<VE>&);

Vector<VE> operator-(const Vector<VE>&,const Vector<VE>&);

Vector<VE> operator*(const Vector<VE>&,double);

Vector<VE> operator*(double,const Vector<VE>&);

Vector<VE> operator/(const Vector<VE>&,double);
```

Figure 3: *Public interface for the class* Vector.

```
double abs(const Vector<Vec2> &v);
double abs(const Vector<Vec3> &v);
```

The dot functions calculate the dot product of two vectors and the abs functions calculate the magnitude of a vector. For the sub-vector template arguments (the two-vector, Vec2, and three-vector, Vec3), the vector is treated as if it were a long scalar array.

The subscript operators both check that their argument is in range. While this slows down access to individual vector elements, it is driven by the philosophy that the Vector class is intended primarily for use in algorithms which operate on the vector as a whole. Small inefficiencies in element access are considered inconsequential and programming safety is preferred.

A remark is needed on the efficiency of the arithmetic functions operator+ and operator-. It would seem that the code

```
Vector<VE> a(n), b(n), c(n);
for (unsigned i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

should be more efficient than the simpler

```
Vector<VE> a(n), b(n), c(n);
c = a + b;
```

In the latter case, the result of a+b must be stored in a temporary vector, then copied to c; in the former case the copying is avoided. In fact, the opposite is true; the simple code is considerably more efficient than the explicit loop. On a long vector of doubles the simple code is about twice as fast. On an SGI Indigo 2, using vectors of a million doubles, the code loop took 2.03 seconds while the simple code took 0.95 seconds. Because the subscript operator is more efficient for a const Vector<VE> (see Section 4.2 for details why), the loop code can be improved somewhat by using const references to a and b:

```
Vector<VE> a(n), b(n), c(n);
const Vector<VE> &ca = a, &cb = b;
for (unsigned i = 0; i < n; i++)
  c[i] = ca[i] + cb[i];
```

This version reduces the time taken to 1.67 seconds, still significantly slower than the simple code. The simple code is faster for two reasons:

1. It avoids the overhead associated with the slow subscript operator functions.
2. Lazy copying is used by both the assignment operator and by the copy constructor so that the result of a+b is, in fact, never copied: that is, the perceived inefficiency described above does not exist. For an explanation of lazy copying, see Section 4.1.

The message should be clear: when possible, use the vector operators provided. If it is necessary to access vector elements directly, use const vectors whenever possible.

The member function discard_values can also be used to enhance efficiency of the Vector operations. A call to discard_values indicates that the current values of the vector elements will not be used again. Calling it just before the vector is stocked with new values can, in

some cases, speed up execution (for an explanation why, see Section 4.2). For example, for a and b both of type `Vector<double>`, the following code ran about 35% faster when the call to `discard_values` was used.

```
a = b;
b.discard_values();
for (i = 0; i < n; i++) b[i] = i;
```

Note, however, that it is never *necessary* to call `discard_values`. Use `discard_values` carefully: always make sure that the vector is re-stocked with values immediately after it is called.

3.2 Sparse Matrices

The sparse matrices described in this memorandum are used in five steps.

1. The matrix is constructed. This sets the dimensions of the matrix.
2. Elements are added to the profile. The values of the elements cannot be used at this time.
3. The profile is 'fixed'. Once the profile has been fixed, no more elements can be added to it. However, the matrix elements can now be used.
4. The matrix is populated: i.e. values are assigned to each of the matrix elements in the profile.
5. The matrix is used.

For example, in the following code a diagonal sparse matrix, `m`, is created, set to the identity, and used to multiply a vector `v`. The matrix elements are stored in compressed row storage (CRS) format for which the appropriate matrix class is `CRSMtx`.

```
CRSMtx<double,double> m(10,10); // Construct the matrix
Vector<double> v(10), w(10);
int i;
for (i = 0; i < 10; i++)   // Define the profile
  m.add_to_profile(i,i);
for (i = 0; i < 10; i++) { // Populate the matrix and vector
  m(i,i) = 1.0;
  v[i] = i;
}
w = m*v; // Use the matrix
```

Notice that it is not necessary to fix the matrix profile explicitly. As soon as the value of any matrix element is specified, the profile will be fixed.

```
template<class ME, class VE>
class SpMtx
{
public:
  unsigned num_rows() const;
  unsigned num_cols() const;
  int is_square() const;

  SpMtx<ME,VE>& operator=(const SpMtx<ME,VE>&);

  void add_to_profile(unsigned i, unsigned j);
  void fix_profile();

  ME& operator()(unsigned i, unsigned j);
  const ME& operator()(unsigned i, unsigned j) const;

  virtual void zero();
  SpMtx<ME,VE>& negate();
  SpMtx<ME,VE>& operator+=(const SpMtx<ME,VE>&);
  SpMtx<ME,VE>& operator-=(const SpMtx<ME,VE>&);
  SpMtx<ME,VE>& operator*=(double);
  SpMtx<ME,VE>& operator/=(double);

  int is_mem() const;
  void get_mem();
  void free_mem();
};

int operator==(const SpMtx<ME,VE>&, const SpMtx<ME,VE>&);

int operator!=(const SpMtx<ME,VE>&, const SpMtx<ME,VE>&);

Vector<VE> operator*(const SpMtx<ME,VE>&, const Vector<VE>&);

Vector<VE> operator*(const Vector<VE>&, const SpMtx<ME,VE>&);

SpMtx<ME,VE> operator-(const SpMtx<ME,VE>&);

SpMtx<ME,VE> operator+(const SpMtx<ME,VE>&,const SpMtx<ME,VE>&);

SpMtx<ME,VE> operator-(const SpMtx<ME,VE>&,const SpMtx<ME,VE>&);

SpMtx<ME,VE> operator*(const SpMtx<ME,VE>&,double);

SpMtx<ME,VE> operator*(double,const SpMtx<ME,VE>&);

SpMtx<ME,VE> operator/(const SpMtx<ME,VE>&,double);
```

Figure 4: *Public interface for the class* SpMtx.

3.2.1 The Sparse Matrix Base Class

The `SpMtx` class serves as a base class for all the sparse matrices in TRANSOM. It is an abstract class whose member functions define the common interface to the different sparse matrix representations. Figure 4 shows the public interface for this class.

`SpMtx` is a parameterized class; the template argument `ME` is the type of each element of the matrix, whereas the type `VE` is the type of each element of the vectors which it can multiply. Thus, in the example above, both the elements of the matrix and of the vector were doubles (`CRSMtx` is a specialization of `SpMtx` having the same template arguments). The declarations

```
CRSMtx<Mtx3,Vec3> m(10,10);
Vector<Vec3> v(10), w(10);
```

would have created a matrix whose elements were `Mtx3`'s (3×3 sub-matrices) and vectors whose elements were `Vec3`'s (sub-vectors of length three).

For any two objects `me1` and `me2` of type `ME`, and objects `ve1` and `ve2` of type `VE`, the following operators must be defined:

1. Assignment: `me1 = me2` and `ve1 = ve2` (no return value is assumed).
2. Logical equivalence: `me1 == me2`; `me1 != me2`; `ve1 == ve2`; and `ve1 != ve2`.
3. Multiplication: `me1*me2` returning a result of type `ME`, and `me1*ve1` and `ve1*me1` returning a result of type `VE`.
4. Addition and subtraction: `me1 + me2` and `me1 - me2` returning a result of type `ME`, and `ve1 + ve2` and `ve1 - ve2` returning a result of type `VE`.
5. Increment and decrement operators: `me1 += me2` and `me1 -= me2` returning a result of type `ME`, and `ve1 += ve2` and `ve1 -= ve2` returning a result of type `VE`.
6. Scaling operators: `me1 *= d` and `me1 /= d` returning a result of type `ME`, and `ve1 *= d` and `ve1 /= d` returning a result of type `VE`, where `d` is a double.
7. Setting to zero: `zero(m1)` returning a type of `ME`, and `zero(ve1)` returning a type of `VE`.

The requirement for the template argument `VE` arises from the use of a compiler which does not support member functions with templates. We wish to support efficient matrix-vector multiplication using only a reference to a base sparse matrix (i.e. an `SpMtx`). This can only be done using virtual member functions which can take advantage of the actual representation of the sparse matrix. Therefore, it is necessary to have a member function whose argument includes the vector type. On compilers which do not support templates in member functions, this requires the second template argument for the class. When compilers supporting recent ANSI standards for C++ become widely available, the sparse matrix classes will be modified to remove the second template argument.

The matrix profile is built using successive calls to the member function `add_to_profile`. The call `add_to_profile(i,j)` adds element (i, j) to the profile. Once the value of any matrix element has been set, the profile is fixed and calls to `add_to_profile` will result in an error.

Thus, calls to add_to_profile must precede any calls to the indexing operators, the assignment operator, or the arithmetic operators.

3.2.2 Access to the Matrix Elements

In the SpMtx class indexing is done using the round bracket operator. Thus if m is a SpMtx, m(i,j) is its $(i,j)^{th}$ element. To mirror normal matrix usage as much as possible, it is desirable that m(i,j) be an l-value for a non-const SpMtx m. Then one may write:

```
m(i,j) = e1;
```

to assign the value of an element. However, if (i,j) is outside the matrix profile, m(i,j) cannot be an l-value since the matrix does not store a value for that element. Moreover, for a const SpMtx, an l-value can never be returned. Hence, the indexing operators work somewhat differently for const and non-const SpMtx's:

- The non-const indexing operator will generate a fatal error if (i,j) is outside the matrix profile.
- The const indexing operator will return zero if (i,j) is outside the profile.

In cases where one wishes to use all elements of the matrix, whether within the profile or not, the matrix should be cast to const SpMtx before using the indexing operator. For example, if m is an SpMtx<double,double>, the following code writes the n^{th} row of m to cout:

```
const SpMtx<double,double> &cm = m;
for(unsigned i = 0; i < cm.num_cols(); i++)
  cout << cm(n,i) << ' ';
cout << endl;
```

Without the use of the const reference cm, a fatal error would occur unless every element on row n was in the matrix profile.

The member function zero() sets the values of all elements to zero. When an application uses a sparse matrix only via the base class SpMtx, it is always best to call zero() to initialize the matrix immediately after all elements have been added to the profile. This is because some profile storage schemes contain elements which may not have been added using add_to_profile: for example, the diagonal elements are always contained in the skyline, sparse skyline, and S2dBlock profiles. The call to zero() ensures that these elements are also given values along with those explicitly used by the application.

The assignment operator can be used to copy the values of another sparse matrix. The profile of the target matrix (the one whose values are being updated) must be a superset of the matrix being copied. This function is not very efficient as the profiles of both the target matrix and the copied matrix must be traversed. However, it does allow matrices using different profile storage schemes to be copied, provided that their profiles are compatible. For example, a CRS matrix could be copied into a sparse skyline matrix. When the matrix storage schemes are the same, much more efficient methods for copying the matrices can be used.

3.2.3 Memory Management

Consider the situation in which a very large sparse matrix is to be used repeatedly within a loop, say to solve a linear system. The matrix profile is static, being calculated before the first iteration. On each execution of the loop new values for the matrix are calculated and the linear system is solved. Further memory-intensive calculations are made before the loop re-executes. After the linear system is solved, it may be necessary to free the memory used by the sparse matrix in order that the subsequent calculations may be performed. However, if the matrix is deleted, then the calculation of the profile will have to be brought inside the loop. If this is a costly operation, that may slow execution significantly.

The `SpMtx` memory management member functions `get_mem`, `free_mem`, and `is_mem` have been provided to handle situations such as these. They control the allocation of memory from the heap for the values of the matrix elements; the matrix profile will persist throughout the lifetime of the matrix itself. The function `free_mem` deletes the memory used to store the values of all the matrix elements. This destroys the current value of the matrix, but not its profile. The function `get_mem` allocates memory for the values of the matrix elements, if none has already been allocated. The function `is_mem` returns true if memory has been allocated for the element values, false otherwise.

It is never necessary to call `get_mem` explicitly. Any `SpMtx` member functions which assign element values will call `get_mem` if necessary before assigning the values. For example, `zero` will allocate memory if none is currently available. Nevertheless, it is good programming practice to call `get_mem` and `free_mem` in pairs so that it is clear when the element values have been allocated.

Member functions which expect element values to exist (e.g. the `const` version of the subscript operator) will generate fatal errors if no memory is available.

3.2.4 Iterators over the Non-zero Elements

An `SpMtxIter` iterates over the non-zero elements of a sparse matrix: its public interface is shown in Figure 5. The order of traverse is set using the second argument to the `SpMtxIter` constructor. It may have the following three values:

`SpMtxProf::standard`:
: The elements are traversed in an unspecified order. The actual order used will maximize the efficiency of the traverse based on the type of matrix profile.

`SpMtxProf::row_order`:
: The elements of row 0 are traversed first by increasing columns, then row 1, then row 2, etc.

`SpMtxProf::column_order`:
: The elements of column 0 are traversed first by increasing rows, then column 1, then column 2, etc.

The standard order is by far the most efficient, both in terms of CPU time and memory used. It should be used if at all possible.

```
template<class ME, class VE>
class BaseSpMtxIter
{
public:
  virtual void reset();
  virtual void next();
  virtual int in_range() const;

  unsigned row() const;
  unsigned column() const;
};

template<class ME, class VE>
class SpMtxIter: public BaseSpMtxIter<ME,VE>
{
public:
  SpMtxIter(SpMtx<ME,VE> &m, SpMtxProf::IterType t = SpMtxProf::standard);
  E& value() const;
};

template<class ME, class VE>
class ConstSpMtxIter: public BaseSpMtxIter<ME,VE>
{
public:
  ConstSpMtxIter(const SpMtx<ME,VE> &m,
                 SpMtxProf::IterType t = SpMtxProf::standard);
  const E& value() const;
};
```

Figure 5: *Public interface for the sparse matrix iterator classes.*

The syntax of an SpMtxIter follows that of all TRANSOM iterators (see Reference 1, Section 2.2). SpMtxIter also has the member functions row(), column(), and value() for determining the location and value of the current matrix element. Thus, the following code prints the location and value of every element in the profile of a SpMtx<double,double>, m, in row order:

```
SpMtxIter<double,double> iter(m,SpMtxProf::row_order);
while(iter.in_range()) {
  cout << '(' << iter.row() << ',' << iter.column() << "): "
       << iter.value() << endl;
  iter.next();
}
```

SpMtxIter requires that memory for the element values of the matrix be allocated at the time that the SpMtxIter is constructed. If not, a fatal error will be generated.

Because the member function value() returns an l-value, it cannot be used by a const

```
template<class ME,class VE>
inline void set_prod_mtx(VE &v1, const ME &m, const VE &v2)
{ v1 = m*v2; }

inline void inc_prod_mtx(VE &v1, const ME &m, const VE &v2)
{ v1 += m*v2; }

inline void dec_prod_mtx(VE &v1, const ME &m, const VE &v2)
{ v1 -= m*v2; }

inline void set_prod_mtx_trans(VE &v1, const ME &m, const VE &v2)
{ v1 = v2*m; }

inline void inc_prod_mtx_trans(VE &v1, const ME &m, const VE &v2)
{ v1 += v2*m; }

inline void dec_prod_mtx_trans(VE &v1, const ME &m, const VE &v2)
{ v1 -= v2*m; }
```

Figure 6: *Arithmetic functions for matrix elements.*

SpMtx. Hence, a separate iterator, ConstSpMtxIter is defined for use by const SpMtx's. It is exactly the same as SpMtxIter except that its member function value() returns a const ME&.

3.2.5 Arithmetic Operators

The SpMtx class has been endowed with a full complement of arithmetic operators. In the case of binary operations such as a+b or a-b, it is not clear which profile storage scheme should be used by the returned matrix. In the current implementation the profile of matrix a is used as the profile of the returned matrix. This has the side effect that if b's profile contains elements which are not in a's then an error will result. This problem is usually not serious as adding two large sparse matrices is a rare operation; adding two large matrices with different profiles is even rarer. The addition and subtraction operations are much more efficient when the two matrices have the same profile; in that case the two value arrays can simply be added or subtracted with no need to worry about the location of the elements in the matrices.

The functions defined in Figure 6 may be used to speed up the matrix arithmetic functions. Suppose that m is of type ME and v1 and v2 are of type VE. If ME is a sub-matrix and VE is a sub-vector, then the assignment v1 = m*v2 will normally require two function calls: one to evaluate m*v2 and another to copy the temporary result to v1. The assignment can often be performed more efficiently using a single function. This can be done by overloading the function set_prod_mtx using the particular instances of ME and VE. For example, in TRANSOM, the following overloaded version of set_prod_mtx is used to generate efficient code when ME is a Mtx3 (a 3×3 matrix of doubles) and VE is a Vec3 (a three-vector of doubles):

```
template<class ME, class VE>
class CRSMtx: public SpMtx<ME,VE>
{
public:
  CRSMtx(unsigned nr, unsigned nc);
  CRSMtx(const CRSMtx<ME,VE> &m);

  CRSMtx<ME,VE>& operator=(const CRSMtx<ME,VE> &m);
  CRSMtx<ME,VE>& operator=(const SpMtx<ME,VE> &m);
};
```

Figure 7: *Public interface for the class* **CRSMtx.**

```
template<class ME, class VE>
class S2dBlockMtx: public SpMtx<ME,VE>
{
public:
  S2dBlockMtx(unsigned n1, unsigned n2);
  S2dBlockMtx(const S2dBlockMtx<ME,VE> &m);

  S2dBlockMtx<ME,VE>& operator=(const S2dBlockMtx<ME,VE>& m);
  S2dBlockMtx<ME,VE>& operator=(const SpMtx<ME,VE> &m);
};
```

Figure 8: *Public interface for the class* **S2dBlockMtx.**

```
inline void set_prod_mtx(Vec3 &v1, const Mtx3 &m, const Vec3 &v2)
{
  v1[0] = m[0][0]*v2[0] + m[0][1]*v2[1] + m[0][2]*v2[2];
  v1[1] = m[1][0]*v2[0] + m[1][1]*v2[1] + m[1][2]*v2[2];
  v1[2] = m[2][0]*v2[0] + m[2][1]*v2[1] + m[2][2]*v2[2];
}
```

The functions in Figure 6 are used whenever possible by the vector and matrix arithmetic functions. The definitions shown in Figure 6 are defaults that ensure that the functions are defined if not overloaded. The defaults are perfectly adequate for the simple floating point element types such as `float` or `double`, or if efficiency is not a major concern.

3.2.6 Specializations of SpMtx

Specializations of SpMtx have been defined which use CRS, skyline, sparse skyline, and S2dBlock storage. These classes are called CRSMtx, SkyMtx, SpSkyMtx, and S2dBlockMtx, respectively. Their public interfaces are shown in Figures 7–10. They each inherit all the public member functions of SpMtx.

```
template<class ME, class VE>
class SpSkyMtx: public SpMtx<ME,VE>
{
  SpSkyMtx(unsigned n, int sym_mtx = 0, int sym_prof = 0);
  SpSkyMtx(const SpSkyMtx<ME,VE> &m);

  SpSkyMtx<ME,VE>& operator=(const SpSkyMtx<ME,VE> &m);
  SpSkyMtx<ME,VE>& operator=(const SpMtx<ME,VE> &m);

  int is_sym_mtx() const;
  int is_sym_prof() const;
  void transpose();
  int is_transposed() const;
};
```

Figure 9: *Public interface for the class* SpSkyMtx.

```
template<class ME, class VE>
class SkyMtx: public SpMtx<ME,VE>
{
  SkyMtx(unsigned n, int sym_mtx = 0, int sym_prof = 0);
  SkyMtx(const SkyMtx<ME,VE> &m);

  SkyMtx<ME,VE>& operator=(const SkyMtx<ME,VE> &m);
  SkyMtx<ME,VE>& operator=(const SpMtx<ME,VE> &m);

  int is_sym_mtx() const;
  int is_sym_prof() const;
  void transpose();
  int is_transposed() const;

  int is_factored() const;

  void zero();
};

int solve(SkyMtxRep<ME,VE> &m, Vector<VE> &x, const Vector<VE> &b);
```

Figure 10: *Public interface for the class* SkyMtx.

Each of these classes has two assignment operators: one from a matrix of the same type, and one from an SpMtx. Both the copy constructor and the assignment from a matrix of the same type use lazy copying (see Section 4.1) to reduce unnecessary copying of the element values. On the other hand, the assignment from a SpMtx uses a full copy of the element values, since the copied matrix may not use the same storage scheme as the target matrix.

The two unsigned arguments to the CRSMtx constructor specify the size of the matrix (recall that CRS storage makes no constraints on the matrix dimensions).

The two unsigned arguments to the S2dBlockMtx constructor specify the size of the block which gives the matrix its structure (see Section 2.1.4). The size of the matrix is the product of these arguments.

The constructors of the SkyMtx and the SpSkyMtx are similar. The first argument specifies the size of the matrix, the second specifies whether the matrix is symmetric, and the third specifies whether the profile is symmetric. If the matrix is symmetric, then the third argument is disregarded since the profile must also be symmetric. These classes also have member functions for determining whether the matrix is symmetric or has a symmetric profile. They may also be transposed.

It is important to note that, if each element is a sub-matrix, the member function transpose will not transpose the elements themselves. For example, suppose that matrix element (i, j) is a 2×2 vector:

$$M_{ij} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{3.1}$$

Then after transposition one has

$$M_{ji} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \tag{3.2}$$

not

$$M_{ji} = \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} \tag{3.3}$$

On the other hand, multiplication by a transposed vector will transpose the individual elements of the vector correctly. This ensures that for matrix m and vector v

```
m*v;
```

yields the same result as

```
m.transpose();
v*m;
```

This behaviour has been chosen to simplify implementation of the linear system solvers which require multiplication by the transpose of the matrix. To avoid confusion it is recommended that use of the transpose operation be avoided when the matrix elements are sub-matrices.

The class SkyMtx provides a service which the other SpMtx specializations lack: a direct linear system solver via the function solve(SkyMtxRep<ME,VE> &m, Vector<VE> &x, const Vector<VE> &b). Gaussian elimination is used, *without pivoting*. This function factors the matrix into LU form, thereby destroying the original values of the matrix elements. If it is necessary to retain the original matrix, it should be copied before solve is called. A second

```
template<class ME, class VE>
class PreSpMtx
{
public:
  PreSpMtx(const SpMtx<ME,VE> &m);
  virtual ~PreSpMtx() { }

  unsigned num_rows() const { return get_mtx()->num_rows(); }
  unsigned num_cols() const { return get_mtx()->num_cols(); }
  int is_square() const     { return num_rows() == num_cols(); }

  virtual void make() = 0;
  virtual int is_mem() const { return 1; }
  virtual void get_mem()  { }
  virtual void free_mem() { }
  void check_mem() const;

};

void solve(const PreSpMtx<ME,VE> &pmtx, Vector<VE> &x, Vector<VE> &b);

void solve(Vector<VE> &x, const PreSpMtx<ME,VE> &pmtx, Vector<VE> &b);
```

Figure 11: *Public interface for the preconditioner classes.*

call to solve will be more efficient than the first since the factorization can be skipped. The member function is_factored can be used to determine whether a matrix has been factored. The member function zero() has been overloaded so that it returns the matrix to an unfactored state.

Because addition and factorization are not commutative (the factorization of the sum of two matrices is not equal to the sum of the factorizations) the addition and subtraction operations on a factored SkyMtx will not work correctly; they will generate errors should they be called when the matrix is factored. On the other hand, the negation, scaling and multiplication operations still work correctly whether the matrix is factored or not.

3.3 Preconditioners

The class PreSpMtx is an abstract base class used to represent the preconditioning matrices M_1 and M_2 described in Section 2.2.1. Its public interface is shown in Figure 11. Each PreSpMtx preconditions a single SpMtx which is specified by the argument, m, of the PreSpMtx constructor. The member function make() calculates the preconditioning matrix using the current values of m. Typically make() will be called whenever m is changed, but if it is known that changes to m are small, it may be more efficient not to recalculate the preconditioner.

The friend functions

```
solve(const PreSpMtx<ME,VE> &m, Vector<VE> &x, Vector<VE> &b);
solve(Vector<VE>&x, const PreSpMtx<ME,VE>&m, Vector<VE>&b);
```

solve the linear systems $\mathbf{M} \cdot \mathbf{x} = \mathbf{b}$ and $\mathbf{x} \cdot \mathbf{M} = \mathbf{b}$, respectively, for $\mathbf{x}$ where $\mathbf{M}$ is the preconditioner.

Like `SpMtx`, the class `PreSpMtx` has memory management member functions `is_mem()`, `get_mem()` and `free_mem()` that allow memory used by the preconditioner to be reclaimed and reused. As no memory is allocated by the base class `PreSpMtx`, the default versions of these functions do nothing.

3.3.1 Class PreDiagSpMtx

TRANSOM provides specializations of `PreSpMtx` which implement diagonal and ILU0 preconditioning of sparse matrices. The class `PreDiagSpMtx` represents a diagonal preconditioner for a `SpMtx`. Its public member functions are the same as those of `PreSpMtx`. `PreDiagSpMtx` maintains an array of inverse diagonal elements to increase the efficiency of the `solve` function. The memory management functions `get_mem()` and `free_mem()` are overloaded to allow the memory for this array to be reused if desired.

The `PreDiagSpMtx` member function `make()` must be able to calculate the reciprocal of the diagonal elements of the matrix `m`; therefore, `PreDiagSpMtx` requires that the function `ME& invert(ME&)` be defined. If `ME` is a floating point value such as a `double`, this function simply calculates the reciprocal; if `ME` is a sub-matrix, it calculates the inverse of the sub-matrix. TRANSOM provides overloaded versions of `invert(ME&)` for `ME` of type `float`, `double`, `Vec2`, or `Vec3`.

`PreDiagSpMtx` is an efficient general implementation of the diagonal preconditioner; hence, it has no specializations corresponding to the different profile storage systems.

3.3.2 Class PreILU0SpMtx:

The class `PreILU0SpMtx` is a generic ILU0 preconditioner for a `SpMtx`. Its public member functions are the same as those of `PreSpMtx`. As described in Section 2.2.1, `PreILU0SpMtx` need only store its diagonal elements. The memory management functions `get_mem()` and `free_mem()` are overloaded to allow the memory for these elements to be reclaimed and reused. `PreILU0SpMtx` also requires that the function `ME& invert(ME&)` be defined.

Unlike `PreDiagSpMtx`, the class `PreILU0SpMtx` is not an efficient implementation of an ILU0 preconditioner. Its member functions `make`, `solve`, and `solve_trans` are useful mainly for debugging more efficient specializations. TRANSOM provides three such specializations: `class PreILU0SpSkyMtx` implements ILU0 preconditioning for an `SpSkyMtx`; `class PreILU0SkyMtx` implements ILU0 preconditioning for an `SkyMtx`; and `class PreILU0S2dBlockMtx` implements ILU0 preconditioning for an `S2dBlockMtx`. Each of these specializations overload `make`, `solve`, and `solve_trans` to make direct use of the profile storage scheme. A specialized ILU0 preconditioner for the class `CRSMtx` has not been provided. Because of the row storage above the diagonal, the CRS storage scheme is not well-suited to the ILU0 calculations. The generic preconditioner `PreILU0SpMtx` can be used with a `CRSMtx`, but it is recommended that a different profile storage scheme be used if ILU0 preconditioning is desired.

3.4 Linear System Solvers

TRANSOM provides classes for solving linear systems of equations, $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$, using any of the iterative methods described in Section 2.2. The following example illustrates the method.

```
// Create the linear system
unsigned n = 100;
SpSkyMtx<double,double> m(n); // Choose sparse skyline storage
Vector<double> x(n), b(n);

< Assign values to m and b >

// Create and calculate a preconditioner
PreILU0SpSkyMtx<double,double> p(m); // Choose ILU0 preconditioning
p.make();

// Choose GMRES as the solution method
SpGMRESSolver<double,double> solver;

// Make a preconditioned solver
SpPreLinSysSolver<double,double> pre_solver(solver);

// Solve with left preconditioning
pre_solver.solve(&p,m,0,x,b);

// Solve with right preconditioning
pre_solver.solve(0,m,&p,x,b);
```

When constructing the linear system, three choices must be made: the profile storage scheme for the matrix, the type of preconditioning, and the solution method for the linear system. In the example above a sparse skyline storage scheme was chosen by making the matrix a `SpSkyMtx`. Similarly, the preconditioning was chosen to be ILU0 by making `p` a `PreILU0SpSkyMtx`. For diagonal preconditioning a `PreDiagSpMtx` would have been used. (Note that for ILU0 preconditioning the type of the preconditioner should correspond to the type of the sparse matrix, but for diagonal preconditioning a `PreDiagSpMtx` can be used regardless of the type of matrix.) The solution method was chosen to be GMRES by making `solver` an `SpGMRESSolver`. Other possibilities would have been `SpSuccSubstSolver` to choose the successive substitution method, `SpConjGradSolver` to choose the Conjugate Gradient method, `SpConjGradSqSolver` to choose the Conjugate Gradient Squared method, or `SpQMRSolver` to choose the Quasi-Minimum Residual method. Each of these classes implements the solution method without preconditioning. In order to solve the system with preconditioning, an `SpPreLinSysSolver` is constructed using `solver` as an argument. `SpPreLinSysSolver` promotes `solver` to a preconditioned solver using the method described in Section 2.2.1. Finally, the system is solved using the `SpPreLinSysSolver` member function `solve`.

The first and third arguments to the `SpPreLinSysSolver` member function `solve` are pointers to the left and right preconditioning matrices, respectively. Thus, in `solve(p1,m,p2,x,b)` the argument `p1` is a pointer to the left preconditioning matrix, $\mathbf{M}_1$. If `p1` is null, then it is assumed that this matrix is the identity: i.e. no left preconditioning is used. Similarly, `p2` is a pointer to the right preconditioning matrix, $\mathbf{M}_2$. The argument `m` is the main matrix $\mathbf{A}$, `x` is the solution vector, and `b` is the right hand side vector.

```
template<. . .>
class LinSysSolver
{ public:
  void set_ref_norm(double);
  void reset_relative_norm();
  void set_max_residual(double);
  void set_acc(double);
  void set_max_number_sweeps(unsigned n);

  void set_iter_report_string(const Str &s);
  void set_res_report_string(const Str &s);
  void set_report(ReportStopTest::ReportTime nr,
  ReportStopTest::ReportTime rr);

};

template<class ME, class VE>
class SpSuccSubstSolver: public LinSysSolver<. . .>
{ public:
  SpSuccSubstSolver(double norm(const V&));
};

template<class ME, class VE>
class SpConjGradSolver: public LinSysSolver<. . .>
{ public:
  SpConjGradSolver(double norm(const V&));
};

template<class ME, class VE>
class SpConjGradSqSolver: public LinSysSolver<. . .>
{ public:
  SpConjGradSqSolver(double norm(const V&));
};

template<class ME, class VE>
class SpGMRESSolver: public LinSysSolver<. . .>
{ public:
  SpGMRESSolver();
  void set_restart(unsigned n);
  void report_inner(int);
};
```

Figure 12: *Public interface for the linear system solver classes.*

The public interface for each of the unpreconditioned solvers is shown in Figure 12. They are all specializations of the base class `LinSysSolver`. It is an abstract class whose member functions define the common interface to the different solution methods. For simplicity, the template arguments for `LinSysSolver` have not been shown: they are described in Section 4.5.

The first five `LinSysSolver` member functions control when the iterations of the solver stop. This is determined according to the values of four parameters: ϵ_{acc}, ϵ_{div}, N_{max}, and R. Their values may be set by the functions `set_acc`, `set_max_residual`, `set_max_number_sweeps`, and `set_ref_norm`, respectively. Their default values are:

$$\epsilon_{acc} = 10^{-4}; \qquad \epsilon_{div} = 10^{4}; \qquad N_{max} = 30; \qquad R = 1.0; \tag{3.4}$$

At the start of each iteration, the solver executes a 'stop test' to determine whether the iteration should continue. At this time the residual, $\mathbf{r} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}$, is known. The stop test performs the following steps:

1. The norm of the residual is determined using the function `norm` specified in the constructors of the linear system solver classes.
2. If $\text{norm}(\mathbf{r})/R < \epsilon_{acc}$, then the solver is deemed to have converged. No more iterations are executed.
3. If $\text{norm}(\mathbf{r})/R > \epsilon_{div}$, then the solver has diverged. No more iterations are executed.
4. If the total number of iterations exceeds N_{max}, then no more iterations are executed.

The member function `reset_relative_norm` sets R to the norm of the current value of the residual.

The class `GMRESSolver` does not have a norm function as an argument to its constructor. This is because the norm of the residual is generated as a by-product of the GMRES algorithm and need not be calculated by a norm function.

The remaining member functions specify how the iterations are reported to standard output. At every iteration, both the iteration number and the current value of $\text{norm}(\mathbf{r})/R$ can be reported to standard out.

`set_iter_report_string(const Str &s);`
Sets the string which precedes the iteration count to `s`. The default string is `"Iteration "`.

`void set_res_report_string(const Str &s);`
Sets the string which precedes the residual norm to `s`. The default string is `"Accuracy = "`.

`void set_report(ReportStopTest::ReportTime nr, ReportStopTest::ReportTime rr);`
Sets the time at which the iteration report and residual reports are written. The arguments `nr` and `rr` may have the following values:

`ReportStopTest::before`: The report is written before the stop test is executed.

`ReportStopTest::after`: The report is written after the stop test is executed, but only if no more iterations are to be performed.

`ReportStopTest::after_always`: The report is written after the stop test, irrespective of whether more iterations will be performed.

`ReportStopTest::none`: No report is written.

The default is that no reports are written.

In the following example the successive substitution method is used to solve a matrix using S2dBlock storage. Left diagonal preconditioning is used. The iterations of the solver will stop when the norm of the residual is less than 10^{-8} times the norm of the left hand side vector b, or after 100 iterations, whichever occurs first. The norm of the residual is reported to standard output after each iteration. The function `abs(Vector<Vec2>&)` is used to calculate the norm of the residual.

```
unsigned n = 10;
S2dBlockMtx<Mtx2,Vec2> m(n,n);
Vector<Vec2> x(n*n), b(n*n);
PreDiagSpMtx<Mtx2,Vec2> p(m);
SpSuccSubstSolver<Mtx2,Vec2> solver(abs);

solver.set_acc(1.0e-08);
solver.set_ref_norm(abs(b));
solver.set_max_number_sweeps(100);
solver.set_report(ReportStopTest::before,ReportStopTest::after_always);

< Assign values to m and b >
p.make();

SpPreLinSysSolver<Mtx2,Vec2> pre_solver(solver);
pre_solver.solve(&p,m,0,x,b);
```

When run, this code will produce something like the following on standard output.

```
Iteration 0  Residual norm =             1
Iteration 1  Residual norm =     0.0117958
Iteration 2  Residual norm =    0.000293649
Iteration 3  Residual norm =    1.63957e-05
Iteration 4  Residual norm =     6.4121e-07
Iteration 5  Residual norm =    4.07109e-08
Iteration 6  Residual norm =    2.26821e-09
```

3.4.1 The SpLinSys Class

The class SpLinSys represents a complete linear system; it stores the matrix, right hand side and solution vectors, and, if required, a preconditioner. The choices of profile storage and preconditioner are directed by reading data from an input file. Therefore, classes which use an SpLinSys are completely isolated from the storage scheme and preconditioning used. Such loose coupling is very beneficial in program design as it minimizes the effects of changes made to the linear system solver.

```
template<class ME,class VE>
class SpLinSys: public OFFSRF
{
public:
  SpLinSys(unsigned n);

  void solve();

  SpMtx<ME,VE>& get_mtx();
  Vector<VE>& get_rhs();
  Vector<VE>& get_soln();

  int is_mem() const;
  void get_mem();
  void free_mem();

  void calc_preconditioner();

  virtual int read_record(OFFSRF_ifstream&, const Str&, int skip = 1);
};
```

Figure 13: *Public interface for the class* SpLinSys.

The public profile for the class SpLinSys is shown in Figure 13. The constructor requires a single argument which specifies the size of the linear system. The member functions get_mtx, get_rhs, and get_soln provide access to the components of the linear system so that the matrix and right hand side vector may be stocked and the solution vector used in subsequent calculations.

SpLinSys assumes that a single preconditioner will be used, either on the right or on the left. Alternatively, no preconditioning can be used. The member function calc_preconditioner causes the preconditioning matrix, if it exists, to be calculated using the current value of the matrix. This allows the application using the SpLinSys to decide how often the preconditioner should be recalculated.

The member functions is_mem, get_mem, and free_mem allow control over the memory allocated for the matrix. They do *not* control the memory allocated for the preconditioner, since, if the preconditioner is not recalculated every time the matrix is changed, it may be necessary for the preconditioner memory to persist longer than that of the matrix.

SpLinSys reads input from a file in OFFSRF format[7]. For this purpose, SpLinSys is derived from the base class OFFSRF. The class OFFSRF_ifstream is an input file stream which can be attached to file in OFFSRF format. All classes derived from the OFFSRF base class have an extractor which will read records from an OFFSRF_ifstream. Thus, if stream is an OFFSRF_ifstream and slv is an SpLinSys, then

```
stream >> slv;
```

will cause slv to read OFFSRF records from the file associated with stream. The following

records are recognized.

```
{MATRIX TYPE: matrix-type }
{PRECONDITIONER: preconditioner-type
  {METHOD: preconditioning-method }
}
{SOLUTION METHOD: solution-method }
```

The specifier *matrix-type* should be one of `sparse-skyline` or `CRS`. The skyline storage scheme is not included as an option due to its large memory requirements; it should only be used when an iterative solver will not suffice. The S2dBlock storage scheme is not included since it is not general. The `S2dSpLinSys` is a specialization of `SpLinSys` which includes S2dBlock storage as an option: see Section 3.4.5. If the MATRIX TYPE record is not included in the input file, sparse skyline storage will be used by default.

The specifier *preconditioner-type* should be one of `left`, `right`, or `none` and the specifier *method* should be either `diagonal` or `ilu0`. If the PRECONDITIONER record is not included in the input file, left ILU0 preconditioning will be used by default.

The specifier *solution-method* should be one of `CGS`, `GMRES`, `QMR`, or `SS`. The Conjugate Gradient method is not included because it requires a symmetric positive definite matrix and because one-sided preconditioning is not appropriate.

In the following example, an `SpLinSys` is created and the solution methods defined by reading from the file `linsysin`.

```
// Construct the linear system
SpLinSys<double,double> slv(100);

// Define slv by reading from linsysin
OFFSRF_ifstream in("linsysin");
in >> slv;

// Stock the linear system
SpMtx<double,double> &mtx = slv.get_mtx();
Vector<double> &b = slv.get_rhs();
< Stock mtx and b using mtx.add_to_profile >

// Solve the linear system
slv.calc_preconditioner();
slv.solve();

// Get the solution
Vector<double> &x = slv.get_soln();
```

3.4.2 Element Reordering

As discussed in Section 2.2.2, for a matrix A of size N, any permutation of the numbers zero through $N-1$ defines a new matrix whose elements have been reordered. The class `Permutation`

```
class Permutation: public Handle
{
public:
  Permutation(unsigned m);
  Permutation& operator=(const Permutation&);

  unsigned len() const;
  unsigned operator[](unsigned i) const;

  void transpose(unsigned i,unsigned j);
  void random();
  void reverse();

  friend Permutation operator*(const Permutation&,const Permutation&);
};
```

Figure 14: *Public interface for the class* `Permutation`.

represents permutations of the numbers zero through $N-1$. Its public interface is shown in Figure 14.

When originally constructed, a `Permutation` is equivalent to the identity permutation. It can be modified using the member functions `transpose`, which transposes two elements, `reverse` which reverses the order of the permutation, and `random`, which creates a random permutation by transposing each element of the permutation with another element chosen at random.

The composition of two permutations p and q is defined by

$$(p * q)_i = p_{q_i} \tag{3.5}$$

The function `operator*(const Permutation&,const Permutation&)` returns the composition of two permutations.

A `PermVector` is a wrapper for a `Vector` so that its elements may be referenced by a permuted index. Thus, if `v` is a `Vector<VE>`, `p` is a permutation, and `pv` is the `PermVector<VE>` created using `p` and `v`, then `pv[n]` is equivalent to `v[p[n]]`. Similarly, a `PermSpMtx` is a wrapper for an `SpMtx` which allows its elements to be referenced by permuted indices. The public interfaces for the classes `PermVector` and `PermSpMtx` are shown in Figure 15.

To solve a linear system with permuted elements, use a `PermVector` and a `PermSpMtx` when adding to the matrix profile, when stocking the elements of the linear system, and when accessing the solution vector. Use the unpermuted vectors and matrix when solving the linear system. In the following example a linear system is solved with the element order reversed:

```
// Define the matrix and vectors
unsigned n = 10;
SpSkyMtx<double,double> m(n,1,1); // Symmetric matrix and profile
Vector<double> x(n), b(n);

// Define the reverse permutation
```

```
template<class VE>
class PermVector
{
public:
  PermVector(Vector<VE> &vin, Permutation &p);

  unsigned len() const;
  void zero();

  VE& operator[](Node i);
  const VE& operator[](Node i) const;
};

template<class ME,class VE>
class PermSpMtx
{
public:
  PermSpMtx(SpMtx<E,VE> &m, Perm &nm);

  unsigned num_rows() const;
  unsigned num_cols() const;
  int is_square() const;
  int is_mem() const;
  void get_mem();
  void free_mem();
  void zero();

  void add_to_profile(Node i, Node j);
  E& operator()(Node i, Node j);
  const E& operator()(Node i, Node j) const;
};
```

Figure 15: *Public interface for the classes* PermVector *and* PermSpMtx.

```
Permutation p(n);
p.reverse();

// Define wrappers for the matrix and vectors
PermSpMtx<double,double> pm(m,p);
PermVector<double> px(x,p), pb(b,p);

// Define the matrix profile: use the wrapper mtx
// Don't need to add the diagonal elements since SpSkyMtx has them by
// default
unsigned i;
for (i = 1; i < n; i++) pm.add_to_profile(i,i-1);

// Stock the matrix and right hand side vector: use wrappers
pb.zero();
pb[n-1] = 1.0;
pm(0,0) = 2.0;
for (i = 1; i < n-1; i++) {
  pm(i,i) = 2.0;
  pm(i,i-1) = 1.0;
}
pm(n-1,n-1) = 2.0;

// Solve the system using unpreconditioned Conjugate Gradient method
// Use unwrapped matrix and vectors
SpConjGradSolver<double,double> solver(abs);
SpPreLinSysSolver<double,double> pre_solver(solver);

pre_solver.solve(0,m,0,x,b);

// Write out the solution: use wrapped vector
for (unsigned j = 0; j < n; j++) {
  cout << px[j] << ' ';
cout << endl;
```

3.4.3 Node Mapping

In TRANSOM, each matrix index, i or j, is associated with a node in a grid. Although each node is numbered, it is not necessarily true that a grid of N nodes will number those nodes using 0 through $N-1$. Therefore it is necessary to have a mapping from the nodes to the matrix indices. This mapping is provided by the class `NodeMap` whose public interface is shown in Figure 16.

When both node mapping and element reordering are used, the node must first be mapped onto an element number, then the element number must be permuted. To avoid the inefficiency associated with this double mapping, `NodeMap` stores a map directly onto the permuted elements. The default element ordering is the natural one (i.e. the permutation used is the identity) but it can be changed using the member functions `re_order`, `reverse`, and `random`.

```
class NodeMap
{
public:
  static const unsigned unmapped;

  NodeMap(const NodeOrg &b);
  void re_order(const Permutation&);
  void reverse();
  void random();
  int is_mapped(Node n);
  unsigned operator[](Node n) const;
};
template<class VE>
class NodeMapVector
{
public:
  NodeMapVector(Vector<VE> &vin, NodeMap &nm);
  int is_mapped(Node n) const;

  unsigned len() const;
  void zero();

  VE& operator[](Node i);
  const VE& operator[](Node i) const;
};

template<class ME,class VE>
class NodeMapSpMtx
{
public:
  NodeMapSpMtx(SpMtx<ME,VE> &m, NodeMap &nm);
  int is_mapped(Node n) const;

  unsigned num_rows() const;
  unsigned num_cols() const;
  int is_square() const;
  int is_mem() const;
  void get_mem();
  void free_mem();
  void zero();

  void add_to_profile(Node i, Node j);
  ME& operator()(Node i, Node j);
  const ME& operator()(Node i, Node j) const;
};
```

Figure 16: *Public interface for the node mapping classes.*

```
template<class ME,class VE>
class NodeMapSpLinSys: public SpLinSys<ME,VE>
{
public:
  NodeMapSpLinSys(const NodeOrg &norg);

  void add_to_profile(Node i, Node j);

  void fix_profile();
  int is_fixed();

  NodeMapSpMtx<ME,VE>& get_mtx();
  NodeMapVector<VE>& get_rhs();
  NodeMapVector<VE>& get_soln();

  const NodeMap& get_node_map() const;

  virtual int read_record(OFFSRF_ifstream&, const Str&, int skip = 1);
};
```

Figure 17: *Public interface for the class* **NodeMapSpLinSys.**

The argument to the NodeMap constructor is a NodeOrg. This class is described in Reference 1, Section 2.3. It represents the organization of the nodes in a grid or block, and defines the numbering system used for the nodes. Each NodeOrg also has an associated iterator which iterates over all the nodes in the the grid or block. The NodeMap constructor defines the initial node mapping using the order in which the nodes are returned by the iterator.

A NodeMapVector is a wrapper for a Vector so that its elements may be referenced by node numbers rather than element indices. Thus, if v is a Vector<VE>, nm is a node map, and nmv is the NodeMapVector<VE> created using nm and v, then nmv[n] is equivalent to v[nm[n]]. Similarly, a NodeMapSpMtx is a wrapper for an SpMtx which allows its elements to be referenced by node numbers. The public interfaces for the classes NodeMapVector and NodeMapSpMtx are shown in Figure 16.

The procedure for solving a node mapped linear system is similar to that used for a linear system with permuted elements: use a NodeMapVector and a NodeMapSpMtx when adding to the matrix profile, when stocking the elements of the linear system, and when accessing the solution vector. Use the unmapped vectors and matrix when solving the linear system. Alternatively, use the class NodeMapSpLinSys described in the next section.

3.4.4 The NodeMapSpLinSys Class

The class NodeMapSpLinSys is a specialization of SpLinSys which allows node mapping and element reordering. Its public interface is shown in Figure 17.

The NodeMapSpLinSys constructor requires a NodeOrg argument. This defines the node mapping used by the linear system (see Section 3.4.3).

```
template<class E,class VE>
class S2dSpLinSys: public NodeMapSpLinSys<E,VE>
{
public:
  S2dSpLinSys(const S2dNodeOrg &b);
};
```

Figure 18: *Public interface for the class* **S2dSpLinSys.**

Like the profile storage scheme, the preconditioning method, and the solution method, the element ordering is determined from a record in an OFFSRF input file. As well as the MATRIX TYPE, PRECONDITIONER, and SOLUTION METHOD records defined by SpLinSys, the record

{NODE ORDERING: *node-order* }

is defined, where *node-order* is one of cheapest, identity, reverse, random, Cuthill-McKee, or reverse Cuthill-McKee. The specifier identity is equivalent to no element re-ordering, reverse reverses the order of the nodes, random chooses a permutation at random, the Cuthill-McKee algorithm[8] is specified by Cuthill-McKee, and reverse Cuthill-McKee is the Cuthill-McKee order reversed. The specifier cheapest chooses the node ordering among identity, reverse, Cuthill-McKee, and reverse Cuthill-McKee which minimizes the mean bandwidth of the matrix.

When the Cuthill-McKee algorithm is used, whether reversed or not, the locations of the elements in the unpermuted profile must be known before the re-ordering can be determined. For this reason the matrix profile is specified by calling the NodeMapSpLinSys member function add_to_profile repeatedly. When all elements have been added, fix_profile is called. This causes the element ordering to be calculated and the profile to be defined by repeated calls to SpMtx<ME,VE>::add_to_profile.

3.4.5 The S2dSpLinSys Class

An S2dSpLinSys is a NodeMapSpLinSys which is suitable for the solution of linear systems derived from partial differential equations on a single structured block. In addition to the CRS and sparse skyline profile storage schemes, it allows S2dBlock storage for the linear system matrix. This is the default but can also be specified explicitly using the following record in the input file:

{MATRIX TYPE: s2dblock }

S2dSpLinSys also allows non-periodic ILU0 preconditioning to be used in conjunction with S2dBlock storage. This is specified using the record:

```
{PRECONDITIONER: left or right }
  {METHOD: np_ilu0 }
}
```

Finally, S2dSpLinSys allows a new type of element re-ordering denoted 'block order'. On an $N \times M$ block, the natural element order increments index i first, then j: i.e. one first has the N elements for $j = 0$, then N more for $j = 2$, and so on. If $N \leq M$, block order is equivalent to the natural order. However, if $N > M$, then block order increments j first followed by i. The result is a matrix whose mean bandwidth is close to the lower of M and N. 'Reverse block order' is obtained by reversing block order. These element orders can be specified using the following records in the input file:

```
{NODE ORDERING: block }
```

or

```
{NODE ORDERING: reverse block }
```

The public interface for S2dSpLinSys, shown in Figure 18, is almost completely inherited from the NodeMapSpLinSys and SpLinSys. The argument to the constructor is a node organizer for a structured two-dimensional block. The class S2dNodeOrg is described in Reference 2.

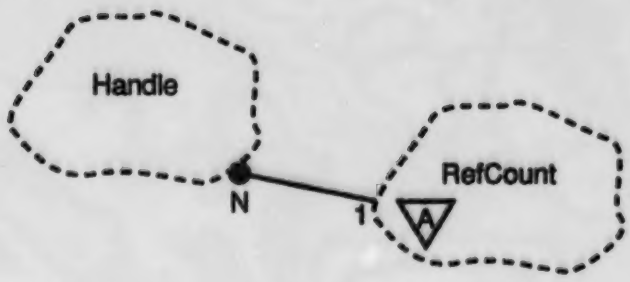

Figure 19: *Class diagram for shared representations.*

4 Implementation of the Sparse Matrix Classes

In this section, implementation of the sparse matrix classes is discussed in detail.

4.1 Shared Representations

Using shared representations for objects (also known as handle classes) is a technique which has several advantages, the most commonly cited being lazy copying and garbage collection (see Reference 9, Chapter 13.9 for a more complete discussion). It has been implemented for the vector and sparse matrix classes primarily for the run-time efficiency obtained by lazy copying.

The basic problem was outlined in Section 3.1; the code

```
Vector<VE> a(n), b(n), c(n);
c = a + b;
```

appears to require a redundant copy of the result into the vector c. The redundant copy can be avoided if one splits the vector object into two parts: a handle which provides the vector interface and a representation which provides the vector state. The handle contains a pointer to the representation. The term 'lazy copying' means that when the handle is copied, only the pointer is copied, not the representation.

Once the pointer to the representation has been copied, two or more objects both share the same representation. The representation must keep count of the number of handles which use it; this is often called the reference count and the representation is called a 'reference counted object'. When the vector is deleted, it decreases the reference count of its representation by one. When the reference count falls to zero, the representation may be deleted, since it is no longer used. Automatic deletion of objects in this manner is called garbage collection.

The classes `Handle` and `RefCount` are abstract base classes for implementing handles and reference counted objects. Their relationship is shown in Figure 19 using Booch's notation[10]. The line with the filled circle indicates that each `Handle` has a `RefCount` as a member, but that one `RefCount` may be shared by many (N) `Handle`'s.

`RefCount` is a mix-in class used only by the `Handle` class and its specializations; hence, none of its members are public. Its interface is shown in Figure 20.

```
class RefCount
{
  friend class Handle;
private:
  int refcnt;

protected:
  RefCount()  refcnt = 0;
  RefCount(const RefCount&)  refcnt = 0;
  virtual ~RefCount()

  virtual RefCount* copy_self() const = 0;
  virtual RefCount* new_rep() const = 0;

private:
  int inc_ref()       return ++refcnt;
  int dec_ref()       return --refcnt;
  int count() const   return refcnt;
};
```

Figure 20: *Interface for the class* **RefCount**.

Note that the abnormal behaviour of the copy constructor (i.e. the argument object is not actually copied) is what is required when new representations are constructed by copying.

The member function `copy_self()` makes a new object and copies the current one to it. A pointer to the new object is returned. The reference count of the new object should be 0. In contrast, the member function `new_rep()` makes a new representation but does not copy the current object to it.

The interface for the `Handle` class is shown in Figure 21. Classes derived from `Handle` should not delete the shared representation, even if they allocate it. The `Handle` class handles this, ensuring that it is not deleted when still in use by another object.

It is assumed that the implementation of a class as a handle with a separate representation is hidden from the user. Therefore the complete `Handle` interface is also hidden from the user; it has no public members.

The copy constructor and the assignment operator both cause the current object to share the representation with the object being copied. The reference count of the representation is increased by one. The destructor decreases the reference count of `ref` and deletes it if the reference count is zero.

Consider the following code:

```
Vector<VE> a(10), b(10);
a = b;
a[0] = 1.0;
```

If the `Vector` class is implemented as a handle, then after the assignment, `a` and `b` share the

```
class Handle
{
protected:
  RefCount *rep;

  Handle(RefCount *r);

  Handle(const Handle &r);
  ~Handle();

  Handle& operator=(const Handle &r);

  void ensure_unique_rep();
  void make_new_rep();
};
```

Figure 21: *Interface for the class* **Handle**.

same representation. The value of the first element of a is then changed to 1.0. If we simply change the value in the shared representation, the value of the first element of b will also change. This is clearly not what is desired in most cases. When the value of a is changed but the value of b is to remain the same, it is necessary to copy the representation before making the change to a. The Handle member function ensure_unique_rep() ensures that the representation of an object is unique. It examines the reference count of rep. If it is not one, the representation is copied and rep is altered to point to the new copy. When the representation is unique it is safe to change the representation without causing side effects to other objects. In general this means that all non-const member functions of classes derived from Handle should call ensure_unique_rep(). In the example, the Vector subscript operator, VE& operator[](unsigned), must call ensure_unique_rep() before returning a reference to the array element.

Consider now how the Vector member function zero() might be implemented. It sets the value of every element in the vector to zero. If the vector shares a representation, it must be made unique before the values are changed. However, ensure_unique_rep() will copy all the existing values of the vector before changing them all to zero. Clearly the copy is unnecessary. Instead, the Handle member function make_new_rep() can be used to ensure that the representation is unique without copying the old one. If the representation is already unique, nothing is done.

The functions ensure_unique_rep() and make_new_rep() are implemented using the member functions copy_self() and new_rep() in RefCount. The exact meaning of 'creation of a new representation without copying' will therefore be defined by the implementation of the virtual function new_rep() in derived classes. In many cases no distinction will be necessary and new_rep() and copy_self() will be identical.

```
template<class E>
class ArrayRep: public RefCount
{
  unsigned length;
  E *vals;

public:
  ArrayRep(unsigned n);
  ArrayRep(const ArrayRep<E>&);
  virtual ~ArrayRep();

  unsigned len() const;

  virtual RefCount* copy_self() const;
  virtual RefCount* new_rep() const;

  E& operator[](unsigned i)              { return vals[i]; }
  const E& operator[](unsigned i) const; { return vals[i]; }

protected:
  operator E *()             { return vals; }
  operator const E *() const { return vals; }

  friend class Vector<E>;
};
```

Figure 22: *Interface for the class* `ArrayRep`.

4.1.1 The ArrayRep Class

An `ArrayRep` is a reference counted representation of an array: i.e. n copies of objects of class `E` at contiguous memory locations. Its interface is shown in Figure 22.

The subscript operators do not check for a unique representation since, in general, this is not always desirable. For example, one may wish to implement a sub-string or sub-vector such that changes to the sub-vector also cause changes to the original vector. Checking for a unique representation is the responsibility of the handle, not the representation.

4.2 Vectors

A vector is an array of numerical values upon which arithmetic operations can be performed. The class `Vector` is an implementation of vectors using a handle and a representation which is an `ArrayRep`: see Figure 23 (the arrows in the figure show that the class at the base of the arrow is a specialization of the class at the tip). Its public interface is shown in Figure 3. A private constructor with the following prototype is also defined:

```
Vector(RefCount*);
```

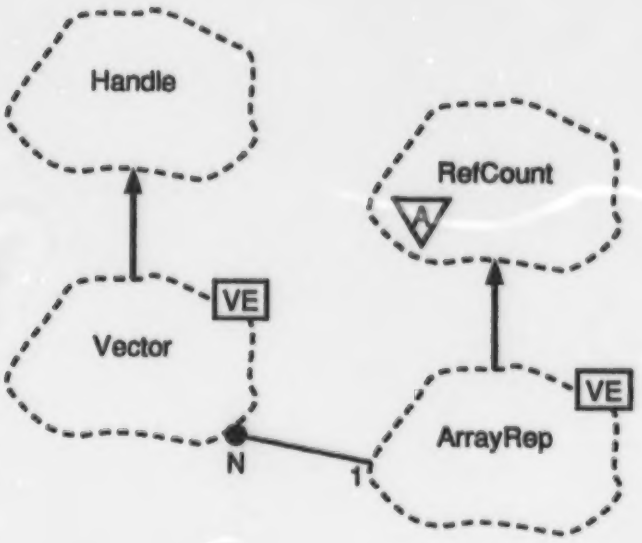

Figure 23: *Class diagram for vectors.*

This function is never defined. It serves only to hide the similar `Handle` constructor so that it cannot be used.

Implementation of `Vector` as a `Handle` has clear performance implications. Access to the vector elements via `VE& operator[] (unsigned)` is slowed by the need to check for a unique representation. When individual access to the vector elements is used a great deal and operations on the vector as a whole comparatively little, then the vector will be best implemented without a handle. For use with sparse matrices and linear system solvers, the efficiency gained by lazy copying normally outweighs the loss of efficiency in stocking the vector. In line with the philosophy that vector access is not a limiting time constraint, the subscript operators also check their arguments to ensure that they are in the correct range.

Note that a `const Vector` need not check for a unique representation (although it does check that the argument is in range). Therefore access to the elements of a `const Vector` is faster than the access to a non-`const Vector`.

The member function `discard_values`, discussed in Section 3.1, is implemented simply as a call to the `Handle` member function `new_rep`. When the reference count of the vector representation is greater than one, this causes a new vector representation to be allocated without copying the current values. The new vector can then be stocked with new values. The efficiency gain occurs because copying of the old values has been avoided. If the subscript operators are used to stock the vector without a call to `discard_values`, then the `Handle` member function `ensure_unique_rep` will be called instead (by the subscript operator); this copies the old values unnecessarily just before they are all replaced.

4.3 Sparse Matrices

When implementing sparse matrices it is important to distinguish between the representation of the matrix profile and the representation of the values stored by the matrix. For example, when two matrices are added together, they will most often have the same profile. In this case it will be most efficient if the profile is lazy-copied while the elements are changed. This can be done if the class `SpMtx` shares two different representations: one for the profile and one for the element values. It is then possible to share the profile representation while retaining a unique representation for the values. Note, however, that if the representation of the values is shared, the profile representation must also be shared, since, in general, the representation of the values will depend upon the representation of the profile (i.e. the order of the elements in the value array will differ according to the type of profile). This suggests that the representation of the values should contain the representation of the profile. Therefore, the base class `SpMtx` has been defined as a handle class whose representation is a `SpMtxRep`. The class `SpMtxRep` contains an array of element values but also inherits a matrix profile from the base class `SpMtxProf`. `SpMtxProf` is in turn a handle class whose representation is given by the base class `SpMtxProfRep`. This layout of classes is illustrated in Figure 24.

Notice that the class `SpMtxRep` is both a handle, because it is a specialization of the handle class `SpMtxProf`, and a reference counted object, because it is the representation of the handle class `SpMtx`. It is this dual role which provides the flexibility in the design and maximizes run-time speed by reducing copying to a minimum.

The classes `CRSMtx`, `S2dBlockMtx`, `SkyMtx`, and `SpSkyMtx` are specializations of `SpMtx` which differ only in that they use different specialization of the class `SpMtxProfRep` as the representations of their profiles.

4.3.1 The Classes SpMtxProf and SpMtxProfRep

The class `SpMtxProf` is a base handle class for sparse matrix profiles; it is derived from the class `Handle`. Since it does not need to know the type of the elements being stored in the matrix, there is no need for it to be a template. The representation of the profile is provided by the class `SpMtxProfRep` which is a specialization of `RefCount`. The interfaces for `SpMtxProfRep` and `SpMtxProf` are shown in Figures 25 and 26.

The values of the elements of a sparse matrix are stored as an array of values whose length is the number of elements in the profile. This array is stored by the class `SpMtxRep`. The principal service of a `SpMtxProfRep` is to provide a mapping from the position of an element in the matrix, (i, j), to the location of its value in the value array. The mapping is defined by the virtual member function `get_val_pos` which returns the position in the value array of the matrix element (i, j). If (i, j) is not in the profile, -1 is returned. The type `ValIndex` is an index into the value array; its representation is simply an `int`.

An `SpMtxProfRep` is used in three stages:

1. It is constructed.
2. Elements are added to the profile using the member function `add_to_profile`.

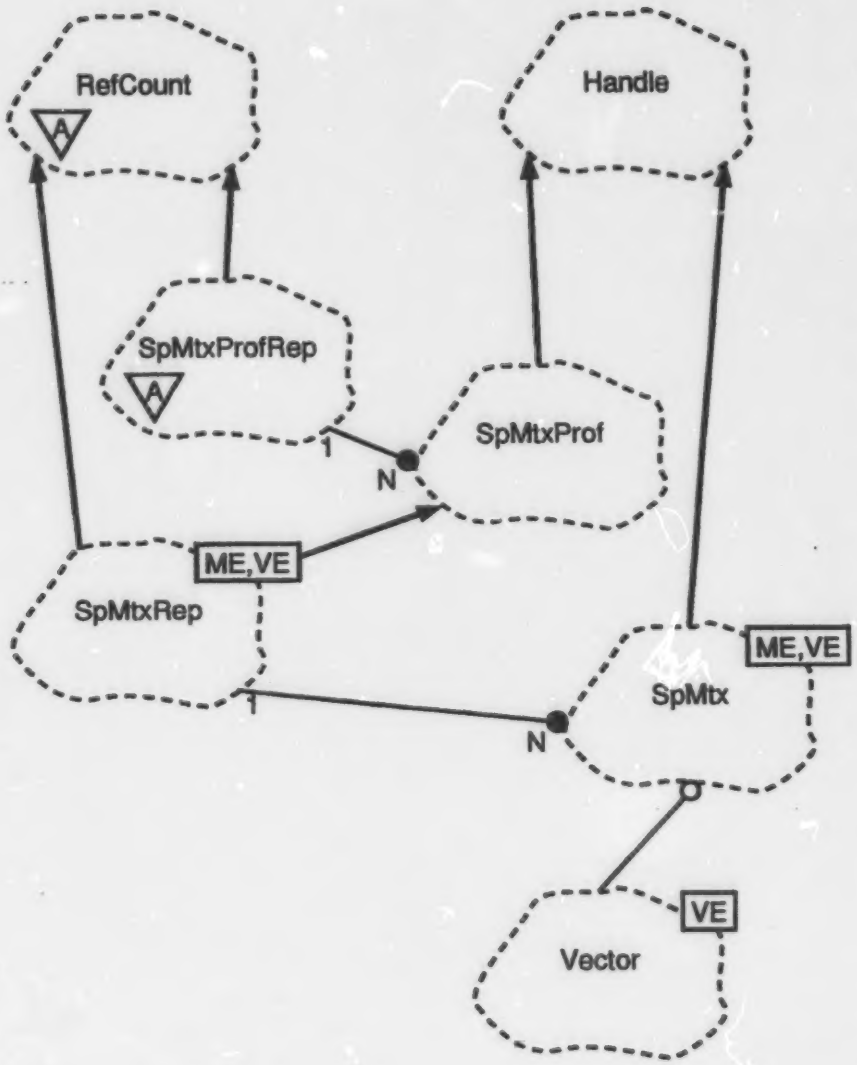

Figure 24: *Class diagram for sparse matrices.*

```
typedef int ValIndex;
class SpMtxProfIter;

class SpMtxProfRep: public RefCount
{
public:
  SpMtxProfRep(unsigned nr, unsigned nc);
  SpMtxProfRep(const SpMtxProfRep&);
  virtual ~SpMtxProfRep()

  unsigned num_rows() const;
  unsigned num_cols() const;

  virtual RefCount* copy_self() const;

  virtual void add_to_profile(unsigned i, unsigned j) = 0;

  virtual void fix_profile();
  int is_fixed() const;

  virtual ValIndex get_val_pos(unsigned i, unsigned j) const = 0;

  virtual unsigned profile_size() const = 0;
  virtual unsigned memory_size() const = 0;

  virtual SpMtxProfIter* get_prof_iter() const = 0;

};
```

Figure 25: *Interface for the class* `SpMtxProfRep`.

3. The profile is used during matrix calculations.

In general, data structures which allow the elements to be added efficiently will not be the same as data structures which allow the representation to be used efficiently. Therefore, `SpMtxProfRep` implements a clear separation between step 2 and step 3. When all the elements have been added to the profile, it is 'fixed' (by calling the member function `fix_profile`). At this time the data structures may be converted. The profile must not be used for matrix calculations before it is fixed; however, once it is fixed, no more elements may be added to the profile.

The size of the profile (i.e. the maximum number of non-zero elements which can be stored in the matrix) is returned by the member function `profile_size`. It is used by `SpMtx` when allocating memory for the value array.

The member function `memory_size` returns the number of bytes of memory required by the profile. For applications which have tight restrictions on memory, it can be used to determine the relative cost in memory of different implementations of the sparse matrix profile.

As is typical of handle classes, `SpMtxProf` does very little, farming out most of its services to its representation.

```
class SpMtxProf: public Handle
{
protected:
  SpMtxProf(SpMtxProfRep *r): Handle(r) { }
public:
  SpMtxProf(const SpMtxProf &p): Handle(p.get_rep()) { }
  virtual ~SpMtxProf() { }

  unsigned num_rows() const   { return get_rep()->num_rows(); }
  unsigned num_cols() const   { return get_rep()->num_cols(); }
  void add_to_profile(unsigned i, unsigned j);
  unsigned profile_size() const { return get_rep()->profile_size(); }

  void fix_profile();
  int is_fixed() const { return get_rep()->is_fixed(); }

  ValIndex get_val_pos(unsigned i, unsigned j) const
    { return get_rep()->get_val_pos(i,j); }

  enum IterType { standard, row_order, column_order };
  virtual SpMtxProfIter* get_prof_iter(IterType t = standard) const;

protected:
  SpMtxProfRep* get_rep() const { return (SpMtxProfRep*)rep; }
};
```

Figure 26: *Interface for the class* SpMtxProf.

4.3.2 The SpMtx and SpMtxRep Class

The interface for the class SpMtxRep, the base class for representations of sparse matrices, is shown in Figure 27. SpMtxRep inherits the matrix profile from SpMtxProf. It is also a specialization of RefCount so that it can act as a representation used by the handle class SpMtx.

The matrix elements are stored at the memory location given by values. The layout of the elements within this memory block is determined by the matrix profile inherited from SpMtxProf. The memory can be allocated and freed using the member functions get_mem and free_mem (see Section 3.2.3).

The RefCount pure virtual function copy_self is defined to create a new SpMtxRep using the copy constructor. It copies all the element values if values is non-null.

The RefCount pure virtual function new_rep is defined to create a new SpMtxRep using the constructor SpMtxRep(SpMtxProfRep*) using the current profile as the argument. It sets values to null (i.e. the matrix elements are not copied) and causes the profile to be shared.

The static member zero_elem is a representation of the zero element of type ME. It is set using the function zero(ME&).

```
template<class ME, class VE>
class SpMtxRep: public RefCount, public SpMtxProf
{
  static ME zero_elem;

protected:
  ME *values;

public:
  SpMtxRep(SpMtxProfRep*);
  SpMtxRep(const SpMtxRep<ME,VE>&);
  virtual ~SpMtxRep() { }

  int is_mem() const;
  virtual void get_mem();
  virtual void free_mem();

  virtual unsigned memory_size() const;

protected:
  virtual RefCount* copy_self() const;
private:
  virtual RefCount* new_rep() const;

public:
  void zero();

  ME& operator()(unsigned i, unsigned j);
  const ME& operator()(unsigned i, unsigned j) const;

  virtual Vector<VE> mult(const Vector<VE>&) const;
  virtual Vector<VE> transpose_mult(const Vector<VE>&) const;

  friend class SpMtxIter<ME,VE>;
  friend class ConstSpMtxIter<ME,VE>;
};
```

Figure 27: *Interface for the class* SpMtxRep.

The indexing operators are implemented using the SpMtxProf virtual function get_val_pos to determine the location of the element in the value array values.

A general version of matrix multiplication is implemented in mult and transpose_mult. Specializations of SpMtxRep overload these functions to provide more efficient versions.

The classes SpMtxIter and ConstSpMtxIter are made friends so that they can get at the matrix elements directly allowing a more efficient implementation of the iterators.

The full interface for the class SpMtx is shown in Figure 28 (the public portion of the interface was shown previously in Figure 4).

The assignment operator does *not* use lazy copying. This is to ensure that the profile storage scheme is not changed by the copy. A specialization of SpMtx such as CRSMtx relies on the fact that its representation is a CRSMtxRep. Hence, none of the public SpMtx member functions must be allowed to change the pointer to the representation.

4.3.3 Sparse Matrix Iterators

Each of the iterators over the sparse matrix elements is implemented using the class SpMtxProfIter whose interface is shown in Figure 29. This iterator returns, in turn, an SpProfIndex for each element of the matrix. The struct SpProfIndex contains a field for the row and column of the element and for the index into the array of matrix values.

SpMtxProfIndex is an abstract class: its member functions reset, next, and in_range must be defined by specializations which make use of the profile storage scheme. The function reset must set the data member curr to the first element in the profile and perform any other necessary initialization. The function next must update curr to the next element in the profile, and the function in_range must return false when the iteration is finished.

SpMtxProfIter has four specializations corresponding to the four profile storage schemes: CRSMtxProfIter, SpSkyMtxProfIter, SkyMtxProfIter, and S2dBlockMtxProfIter,

In general, an SpMtxProfIter makes no guarantees about the order in which the matrix elements are traversed. Usually the most efficient order of traverse will depend on the details of the storage scheme. However, there are two specializations of SpMtxProfIter which do guarantee an order of traverse; they are also independent of the profile storage scheme. These are SpMtxProfRowIter, which returns each element in row one, then row two, and so on; and SpMtxProfColIter which returns each element in column one, then in column two, and so on.

An SpMtxProfRowIter is implemented by constructing an array of sorted linked lists of SpProfIndex's. Each element of the array corresponds to one row of the matrix. When the iterator is constructed the array is created and stocked by iterating over all elements of the matrix, inserting each element into the sorted linked list corresponding to the row of the element. Once the array has been stocked, the SpMtxProfRowIter simply iterates through each linked list on each row in turn. The SpMtxProfColIter is implemented in similar fashion.

Clearly SpMtxProfRowIter and SpMtxProfColIter are inherently inefficient both in time, because an extra iteration over the profile is required to create the linked lists as well as the time to insert the elements into them, and in memory, since the array of linked lists must be

```
template<class ME, class VE>
class SpMtx: public Handle
{
protected:
  SpMtx(SpMtxRep<ME,VE> *r);
public:
  virtual ~SpMtx() { }

  unsigned num_rows() const { return get_rep()->num_rows(); }
  unsigned num_cols() const { return get_rep()->num_cols(); }
  int is_square() const     { return num_rows() == num_cols(); }
  int is_mem() const { return get_rep()->is_mem(); }
  void get_mem()     { get_rep()->get_mem(); }
  void free_mem()    { get_rep()->free_mem(); }

  SpMtx<ME,VE>& operator=(const SpMtx<ME,VE>&);

  void add_to_profile(unsigned i, unsigned j)
                    { get_rep()->add_to_profile(i,j); }
  void fix_profile() { get_rep()->fix_profile(); }

  ME& operator()(unsigned i, unsigned j);
  const ME& operator()(unsigned i, unsigned j) const;

  void zero();
  SpMtx<ME,VE>& negate();
  SpMtx<ME,VE>& operator+=(const SpMtx<ME,VE>&);
  SpMtx<ME,VE>& operator-=(const SpMtx<ME,VE>&);
  SpMtx<ME,VE>& operator*=(double);
  SpMtx<ME,VE>& operator/=(double);

  friend Vector<VE> operator*(const SpMtx<ME,VE> &m, const Vector<VE> &v);
  friend Vector<VE> operator*(const Vector<VE> &v, const SpMtx<ME,VE> &m);

protected:
  SpMtxRep<ME,VE>* get_rep() const { return (SpMtxRep<ME,VE>*)rep; }
  friend class PreSpMtx<ME,VE>;
  friend class BaseSpMtxIter<ME,VE>;
  friend class SpMtxIter<ME,VE>;
  friend class ConstSpMtxIter<ME,VE>;
};

int operator==(const SpMtx<ME,VE>&, const SpMtx<ME,VE>&);
int operator!=(const SpMtx<ME,VE> &m1, const SpMtx<ME,VE> &m2);
```

Figure 28: *Interface for the class* SpMtx.

```
struct SpProfIndex
{
  ValIndex index;
  unsigned row, col;
};

class SpMtxProfIter
{
protected:
  SpProfIndex curr;

public:
  SpMtxProfIter() { }
  virtual ~SpMtxProfIter()

  virtual void reset() = 0;
  virtual const SpProfIndex& next() = 0;
  virtual int in_range() const = 0;
  unsigned row() const    { return curr.row; }
  unsigned column() const { return curr.col; }
  ValIndex index() const  { return curr.index; }
};
```

Figure 29: *Interface for the class* `SpMtxProfIter`.

maintained. They should only be used when efficiency is not a concern. A typical use is to write the values of a matrix into a file in a predetermined order.

The iterators `SpMtxIter` and `ConstSpMtxIter` described in Section 3.2.4 are implemented using an `SpMtxProfIter`. The `SpMtxProfIter` is used to iterate over the profile and the `ValIndex` returned at each iteration step is used to retrieve the value of the matrix element. The common features of `SpMtxIter` and `ConstSpMtxIter` are encapsulated in the base class `BaseSpMtxIter`. See Figure 30 for the important features of the interface and implementation.

Copying a `BaseSpMtxIter`, and hence a `SpMtxIter` or a `ConstSpMtxIter`, is not allowed. Since `BaseSpMtxIter` cannot tell the full derived class to which the `SpMtxProfIter` pointer points, the profile iterator cannot be copied. Copying the pointer alone would not have the desired result: when the original iterator was changed, the copy would also change; moreover, the profile iterator would be deleted twice causing a memory management error.

4.3.4 The CRS Matrix

The class `CRSMtx` is a specialization of `SpMtx` which uses compressed row storage. Its representation is a `CRSMtxRep`, a specialization of `SpMtxRep` which has a `CRSMtxProfRep` to represent its profile. The class `CRSMtxProfRep` is a specialization of the class `SpMtxProfRep`. This class structure is illustrated in Figures 24 and 31. The `CRSMtx` interface is shown in Figures 7

```
template<class ME, class VE>
class BaseSpMtxIter
{
protected:
  SpMtxProfIter *prof_iter;

  BaseSpMtxIter(const SpMtx<ME,VE> &m,
                SpMtxProf::IterType t = SpMtxProf::standard)
    { prof_iter = m.get_rep()->get_prof_iter(t); }
public:
  ~BaseSpMtxIter()          { delete prof_iter; }

  void reset()              { prof_iter->reset(); }
  void next()               { prof_iter->next(); }
  int in_range() const      { return prof_iter->in_range(); }
  unsigned row() const      { return prof_iter->row(); }
  unsigned column() const { return prof_iter->column(); }

private:  // prevent copying
  void operator=(const BaseSpMtxIter<ME,VE>&) { }
  BaseSpMtxIter(const BaseSpMtxIter<ME,VE>&) { }
};

template<class ME, class VE>
class SpMtxIter: public BaseSpMtxIter<ME,VE>
{
  ME *values;
public:
  SpMtxIter(SpMtx<ME,VE> &m, SpMtxProf::IterType t = SpMtxProf::standard)
    : BaseSpMtxIter<ME,VE>(m,t) { values = m.get_rep()->values; }
  ME& value() const { return values[prof_iter->index()]; }
};

template<class ME, class VE>
class ConstSpMtxIter: public BaseSpMtxIter<ME,VE>
{
  const ME *values;
public:
  ConstSpMtxIter(const SpMtx<ME,VE> &m,
                 SpMtxProf::IterType t = SpMtxProf::standard)
    : BaseSpMtxIter<ME,VE>(m,t) { values = m.get_rep()->values; }
  const ME& value() const  return values[prof_iter->index()];
};
```

Figure 30: *Interface and implementation of the classes* `SpMtxIter` *and* `ConstSpMtxIter`.

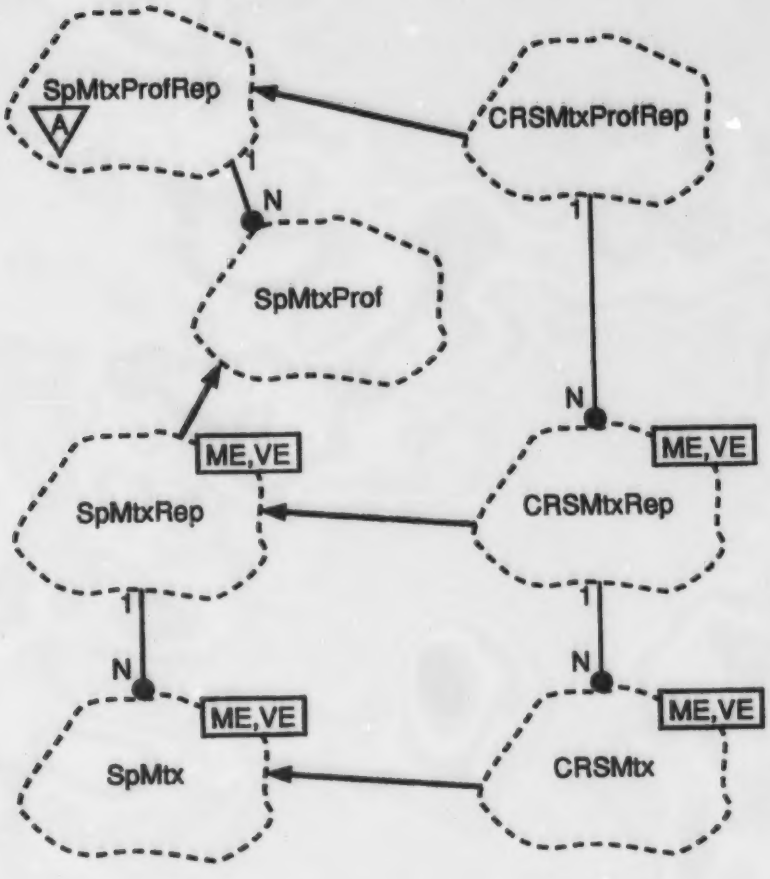

Figure 31: *Class diagram for the CRS matrix.*

and 4.

The interfaces for the classes used to implement `CRSMtx` are shown in Figure 32. `CRSMtxRep` performs two main functions: it ensures that the profile representation is a `CRSMtxProf` and it implements efficient version of matrix multiplication. It is a friend of the class `CRSMtxProfRep` so that the `CRSMtxProfRep` data structures can be used by the member functions `mult` and `transpose_mult`.

`CRSMtxProfRep` implements the CRS profile storage scheme as described in Section 2.1.1. The data members `cols` and `lrow_start` mimic those described in Section 2.1.1. However, these two arrays are not suited for inserting elements into the profile via the member function `add_to_profile`. Therefore, before the profile is fixed, a separate data member, `element_list`, is used to define the profile. It is a list of the locations of each element in the profile, implemented as an array of linked lists of element indices; there is a linked list for every row of the matrix. The indices in the lists are sorted into ascending order. Matrix elements can be inserted into

```
class CRSMtxProfRep: public SpMtxProfRep
{
  Array<ElementList> *element_list;
  unsigned *cols, *row_start;

public:
  CRSMtxProfRep(unsigned nr, unsigned nc);
  CRSMtxProfRep(const CRSMtxProfRep&);
  virtual ~CRSMtxProfRep();

  virtual unsigned profile_size() const;
  virtual void add_to_profile(unsigned i, unsigned j);
  virtual void fix_profile();
  ValIndex get_val_pos(unsigned i, unsigned j) const;
  virtual unsigned memory_size() const;
  SpMtxProfIter* get_prof_iter() const;

protected:
  virtual RefCount* copy_self() const;
  virtual RefCount* new_rep() const;

  friend class CRSMtxProf;
  friend class CRSMtxProfIter;
};

template<class ME, class VE>
class CRSMtxRep: public SpMtxRep<ME,VE>
{
public:
  CRSMtxRep(unsigned nr, unsigned nc)
    : SpMtxRep<ME,VE>(new CRSMtxProfRep(nr,nc))

  CRSMtxRep(const CRSMtxRep<ME,VE> &r): SpMtxRep<ME,VE>(r)
  virtual ~CRSMtxRep()

  virtual Vector<VE> mult(const Vector<VE>&) const;
  virtual Vector<VE> transpose_mult(const Vector<VE>&) const;
};
```

Figure 32: *Interface for the class* CRSMtxRep.

element_list efficiently. When the profile is fixed, element_list is used to create and assign values to cols and lrow_start; it is then deleted.

Iteration over the CRS profile is provided by the class CRSMtxProfIter, a specialization of SpMtxProfIter. Its interface is the same as that for SpMtxProfIter (see Figure 29) except that its constructor has the following prototype:

```
CRSMtxProfIter(const CRSMtxProfRep &p);
```

A pointer to a CRSMtxProfIter is returned by the the CRSMtxProf virtual member function get_prof_iter. CRSMtxProfRep stores a pointer to the representation of the profile being iterated over. It is a friend of the class CRSMtxProfRep so that the iteration can be implemented efficiently using the CRSMtxProfRep data structures.

4.3.5 The Skyline Matrix

The class structure of SkyMtx, SpSkyMtx and S2dBlockMtx are very similar to that of CRSMtx so their interfaces and class diagrams will not be shown. The representation of a SkyMtx is a SkyMtxRep, a specialization of SpMtxRep which ensures that the representation of the profile is a SkyMtxProfRep and which implements efficient versions of matrix multiplication. Iteration over the sparse matrix profile is provided by the class SkyMtxProfIter.

SkyMtxProfRep implements the skyline profile storage scheme as described in Section 2.1.2. The array of element values is segmented into three parts: the diagonal elements, the elements below the diagonal, and the elements above the diagonal. These correspond to the arrays diag_vals, row_vals, and col_vals described in Section 2.1.2. The number of elements below the diagonal is stored in nlower and the number above in nupper. The data member loffset stores the position in the value array of the first element of the sub-array row_vals; similarly, uoffset stores the position of the first element of col_vals. Two additional arrays, ucol_start and lrow_start, exactly mimic those described in Section 2.1.2.

The matrix can be transposed simply by swapping nlower and nupper, loffset and uoffset, and pointers to the arrays lrow_start and ucol_start.

Before the profile is fixed, it is only necessary to store the maximum height of each column above the diagonal, and the maximum length of each row below the diagonal. These values are stored in the arrays lrow_start and ucol_start. When the profile is fixed, these arrays are converted so that their values have the meanings described above.

4.3.6 The Sparse Skyline Matrix

The implementation of the sparse skyline storage scheme is very similar to that of the skyline storage scheme. It has a SpSkyMtxRep for its representation which in turn has a SpSkyMtxRep as the representation of its profile. Iteration over the sparse matrix profile is provided by the class SpSkyMtxProfIter.

SpSkyMtxProfRep has two extra members which the SpKyProfRep does not have: the arrays rows and cols described in Section 2.1.3.

Before the profile is fixed, SpSkyMtxProfRep uses two arrays of ordered linked lists of nodes, each similar to the one used by CRSMtxProfRep. One array is used for the elements above the diagonal, the other for the elements below the diagonal. Both arrays are deleted when the profile is fixed.

4.3.7 The S2dBlock Matrix

The class S2dBlockMtx has an S2dBlockMtxRep for its representation which in turn has a S2dBlockMtxRep as the representation of its profile. Iteration over the sparse matrix profile is provided by the class S2dBlockMtxProfIter.

S2dBlockMtxProfRep implements the S2dBlock profile storage scheme as described in Section 2.1.4. The array of element values is segmented into five parts containing the sub-arrays c1band, b1band, d1band, a1band, e1band, in that order.

4.4 Preconditioning Matrices

This section describes classes which represent the preconditioning matrices M_1 and M_2 discussed in Section 2.2.1. The class PreSpMtx is an abstract base class for preconditioners of sparse matrices of type SpMtx. Its public interface is shown in Figure 11. Each PreSpMtx preconditions a single SpMtx which is specified by the argument, m, of the PreSpMtx constructor. The member function make() calculates the preconditioning matrix using the current values of m. Typically make() will be called whenever m is changed, but if it is known that changes to m are small, it may be more efficient not to recalculate the preconditioner.

PreSpMtx has two protected member functions with the following prototypes:

```
virtual void solve(Vector<VE> &x, Vector<VE> &b) const = 0;
virtual void solve_trans(Vector<VE> &x, Vector<VE> &b) const = 0;
```

They solve the linear systems $\mathbf{M} \cdot \mathbf{x} = \mathbf{b}$ and $\mathbf{x} \cdot \mathbf{M} = \mathbf{b}$ for $\mathbf{x}$, where $\mathbf{M}$ is the preconditioner. These functions are used to implement the two PreSpMtx solve functions:

```
void solve(const PreSpMtx<ME,VE> &pmtx, Vector<VE> &x, Vector<VE> &b);
void solve(Vector<VE> &x, const PreSpMtx<ME,VE> &pmtx, Vector<VE> &b);
```

Because the member functions are virtual, the solve functions will always use the efficient specialized versions of the preconditioning matrix solvers.

Like SpMtx, the class PreSpMtx has memory management member functions is_mem(), get_mem() and free_mem() that allow memory used by the preconditioner to be reclaimed and reused. As no memory is allocated by the base class PreSpMtx, the default versions of these functions are no-ops.

In its first implementation, PreSpMtx shared the matrix representation with the matrix to be preconditioned. This ensured that the matrix representation would not be deleted before the preconditioner. However, it also meant that the matrix would be copied if it were changed between calls to make(). Since this is *not* what is desired, the implementation was changed

so that PreSpMtx simply stores a pointer to the matrix to be preconditioned. It is up to the programmer to ensure that the matrix persists longer than the preconditioner.

As with SpMtx, it would be desirable if PreSpMtx had only one template argument, ME. However, to achieve the polymorphic behaviour of the solve functions, they must be implemented using virtual member functions (solve and solve_trans). With a compiler which does not support template member functions this requires the second template argument. When compilers supporting template member functions are widely available, the template argument VE will be removed.

The implementation of the class PreDiagSpMtx is straightforward and requires no extra description beyond that given in Section 3.3.1.

4.4.1 ILU0 Preconditioning Matrices

The class PreILU0SpMtx is a generic ILU0 preconditioner for a SpMtx. As described in Section 2.2.1, PreILU0SpMtx need only store its diagonal elements. The memory management functions get_mem() and free_mem() are overloaded to allow the memory for these elements to be reclaimed and reused. PreILU0SpMtx also requires that the function ME& invert(ME&) be defined.

The class PreILU0SpMtx is not an efficient implementation of an ILU0 preconditioner. Its member functions make, solve, and solve_trans are useful mainly for debugging more efficient specializations. TRANSOM provides three such specializations: class PreILU0SpSkyMtx implements ILU0 preconditioning for a SpSkyMtx; class PreILU0SkyMtx implements ILU0 preconditioning for a SkyMtx; and class PreILU0S2dBlockMtx implements ILU0 preconditioning for an S2dBlockMtx. Each of these specializations overload make, solve, and solve_trans to make direct use of the profile storage scheme. A specialized ILU0 preconditioner for the class CRSMtx has not been provided. Because of the row storage above the diagonal, the CRS storage scheme is not well-suited to the ILU0 calculations. The generic preconditioner PreILU0SpMtx can be used with a CRSMtx, but it is recommended that a different profile storage scheme be used if ILU0 preconditioning is desired.

4.5 Linear System Solvers

As described in Section 3.4, the base class for all linear system solvers is LinSysSolver. However, the class hierarchy shown in Figure 12 is a little white lie. In fact, none of the member functions shown there actually belong to the class LinSysSolver; they belong to the class ResLinSysSolver intermediate in the hierarchy between LinSysSolver and its specializations. Figure 33 shows a class diagram for LinSysSolver which illustrates this relationship.

LinSysSolver is derived from the base class Solver, described in Reference 1, Section 4.3, which represents an iterative solution method. Every Solver has a Sweep to execute one loop of the iteration, and a StopTest to decide when the iterations should stop. The sweep used by LinSysSolver is a LinSysSweep, a specialization of Sweep which stores pointers to the elements of the linear system: a matrix, a right hand side vector, and a solution vector. The virtual function sweep, which executes a single iteration of the solution method, is not defined

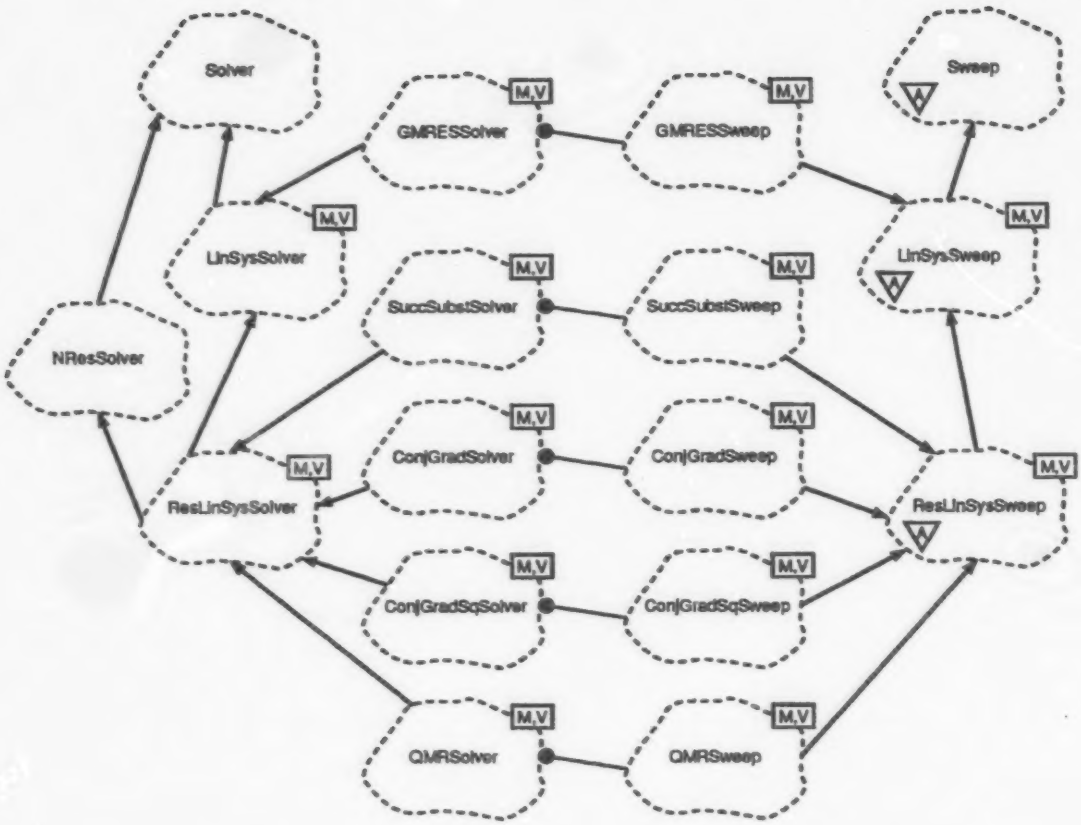

Figure 33: *Class diagram for linear system solvers without preconditioning.*

```
template<class M, class V>
class LinSysSweep: public Sweep
{
  friend class LinSysSolver<M,V>;
protected:
  const M *lhs;
  const V *rhs;
  V *soln;

  LinSysSweep() { lhs = 0; rhs = 0; soln = 0; }
  virtual ~LinSysSweep() { }

  virtual void sweep() = 0;
  virtual void initialize();
  void check_sizes() const;

  void set_lin_sys(const M &m, V &x, const V &b);
};

template<class M, class V>
class LinSysSolver: virtual public Solver
{
protected:
  LinSysSolver(LinSysSweep<M,V> *swp, StopTest *stop);
public:
  virtual ~LinSysSolver() { }

  virtual void solve(const M &m, V &x, const V &b);
private:
  virtual void solve() { Solver::solve(); }
};
```

Figure 34: *Interfaces for the classes* `LinSysSweep` *and* `LinSysSolver`.

by `LinSysSolver`; it is defined by the specializations which implement particular solution algorithms such as Conjugate Gradient or GMRES. The full interfaces for `LinSysSolver` and `LinSysSweep` (no white lies this time) are shown in Figure 34.

It can be seen from these interfaces that `LinSysSolver` provides two main services: it defines the principal data members required by the linear system solver, and it modifies the `Solver` member function `solve` to include arguments which define those members.

`LinSysSolver` and `LinSysSweep` are parameterized classes. The template argument `M` is the type of the matrix, **A**, in the linear system; it must have member functions `num_rows()` and `num_cols` to return the dimension of the matrix. The template argument `V` is the type of the vectors **x** and **b**; it must have a member function `len()` to return the length of the vector. These functions are used to check the dimensions of the elements of the linear system to ensure

that they have consistent sizes.

In addition, if m is of class M, v1 and v2 are of class V, and d is a double, then the following operations must be defined:

1. m*v1 (returns a V).
2. v1 *= d (returns a V).
3. dot(v1,v2) (returns a double equal to the dot product of v1 and v2).
4. abs(v1) (returns a double equal to the magnitude of v1).

In fact, none of these operations is required by LinSysSolver itself, but they are all required by the specializations of LinSysSolver implemented by TRANSOM.

At any stage of an iterative solution of a linear system, the residual vector is defined by $\mathbf{r} = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}$, where x is the current approximation of the solution. The class ResLinSysSolver represents a linear system solver which stores the residual vector and operates by attempting to reduce its magnitude at each iteration. With the exception of GMRES, all the solution methods implemented in TRANSOM are of this type. The interface for ResLinSysSolver is shown in Figure 35. ResLinSysSolver provides three basic services in addition to those inherited from LinSysSolver: it provides a data member to represent the residual vector, it provides a stop test which stops if the residual is either too large or too small or if too many iterations have been executed, and it provides a member function which calculates the residual.

The second of these services is provided by the base class NResSolver. It is a solver with an arbitrary sweep and an NResStopTest. This stop test provides exactly the services required by ResLinSysSolver; it stops either when too many iterations have been completed, or when the norm of the residual is too large (divergence) or sufficiently small (convergence). A function for calculating the norm of the residual is passed as an argument to its constructor. The full interface for NResSolver is shown in Figure 36. It can be seen that the member functions for manipulating the stop test, which were ascribed to LinSysSolver in Section 3.4, are actually members of NResSolver.

The calculation of the residual is handled by the class ResLinSysSweep, a specialization of LinSysSweep which serves as the sweep for ResLinSysSolver.

ResLinSysSolver maintains the vector of residuals, ensuring that it is created, destroyed, and shared by the sweep and the stop test. The latter task is accomplished via the member function set_residual of both ResLinSysSweep and NResSolver. The residual vector is allocated when the solver is initialized and destroyed when the solution is complete. Thus, no memory is wasted while the solver is not actually solving the linear system.

The classes SuccSubstSolver, ConjGradSolver, ConjGradSqSolver, and QMRSolver are all specializations of ResLinSysSolver using the successive substitution, Conjugate Gradient, Conjugate Gradient Squared, and Quasi-Minimum Residual methods, respectively. Each solver has an associated sweep which overloads the ResLinSysSweep virtual function sweep to execute a single iteration of the appropriate algorithm. None of these solvers use preconditioning.

```
template<class M, class V> class ResLinSysSolver;

template<class M, class V>
class ResLinSysSweep: public LinSysSweep<M,V>
{
  friend class ResLinSysSolver<M,V>;

protected:
  V *residual;

  ResLinSysSweep() { residual = 0; }
  virtual ~ResLinSysSweep() { }

  virtual void initialize();
  virtual void sweep() = 0;

  virtual void calc_residual();
  void set_residual(V *r) { residual = r; }
};

class ResLinSysSolver: public LinSysSolver<M,V>, public NResSolver<V>
{
  ResLinSysSweep<M,V> *res_sweep;

public:
  ResLinSysSolver(ResLinSysSweep<M,V> *s, double nrm(const V&));
  virtual ~ResLinSysSolver() { }

  virtual void solve(const M &m, V &x, const V &b);
};
```

Figure 35: *Interfaces for residual linear system solver classes.*

```
template<class R>
class NResSolver: virtual public Solver
{
protected:
  NResStopTest<R> nrst;

public:
  NResSolver(Sweep *s, R *resid, double norm(const R&), double refnorm);
  virtual ~NResSolver() { }

protected:
  void set_residual(R *resid) { nrst.set_residual(resid); }

public:
  void set_ref_norm(double);
  void reset_relative_norm();
  void set_max_residual(double);
  void set_acc(double);
  void set_max_number_sweeps(unsigned);
  void set_iter_report_string(const Str&);
  void set_res_report_string(const Str&);
  void set_report(ReportStopTest::ReportTime,ReportStopTest::ReportTime);
}
```

Figure 36: *Interface for the class* **NResSolver**.

In the following example, a linear system is solved using the Conjugate Gradient Squared method. A sparse matrix using compressed row storage is used for the matrix. Each element of the matrix is a 3×3 sub-matrix and each element of the vectors is a sub-vector of length 3.

```
// Allocate the linear system
unsigned n = 100;
CRSMtx<Mtx3,Vec3> m(n,n);
Vector<Vec3> x(n), b(n);

< Assign values to m and b>

// Make a CGS solver.  Use abs(Vector<Vec3>&) to calculate the norm
ConjGradSqSolver< SpMtx<Mtx3,Vec3>,Vector<Vec3> > solver(abs);

// Solve the linear system
solver.solve(m,x,b);
```

Note that the first template argument of `ConjGradSqSolver` can be `SpMtx<Mtx3,Vec3>` rather than `CRSMtx<Mtx3,Vec3>`. Because `SpMtx` uses virtual functions to implement matrix multiplication, the specialized multiplication routines provided by `CRSMtx` will be used by this solver. Thus, there is no loss of efficiency in using the base class. Moreover, use of the base class has two significant advantages: it prevents unnecessary code instantiations by the compiler, and

```
template<class LP, class M, class RP, class V>
class PreMtx
{
protected:
  const LP *p1;
  const M &mtx;
  const RP *p2;

public:
  PreMtx(const LP *p1in, const M &m, const RP *p2in)
    : p1(p1in), mtx(m), p2(p2in) { }
  unsigned num_rows() const { return mtx.num_rows(); }
  unsigned num_cols() const { return mtx.num_rows(); }

  friend V operator*(const PreMtx<LP,M,RP,V>&, const V&);
  friend V operator*(const V&, const PreMtx<LP,M,RP,V>&);
};
```

Figure 37: *Interface for the class* `PreMtx`.

it allows the solver to be reused to solve another linear system using a different storage scheme for its matrix.

The GMRES solver is unlike the other linear system solvers in that it uses two nested iterations. It is therefore implemented as a `Solver` within a `Solver`. The inner solver performs the bulk of the work, constructing an orthogonal basis for the Krylov sequence $\{\mathbf{A}^k\mathbf{r}_0\}$, where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A} \cdot \mathbf{x}_0$, and $\mathbf{x}_0$ is the initial guess at a solution (see Reference 5 for more details). The inner solver is implemented as a specialization of `LinSysSolver`. The outer solver simply restarts the inner solver using the current approximate solution, so that a new Krylov sequence is generated.

The GMRES method is also different in that the norm of the residual is determined as a by-product of the solution method. Therefore, a vector of residuals need not be maintained and `GMRESSolver` need not be a specialization of `ResLinSysSolver`. Instead, it inherits directly from `LinSysSweep` and `NResSolver<double>` (the second inheritance relation is not shown in Figure 12). As with the other solvers, `NResSolver<double>` provides a stop test and member functions for changing the stopping criterion. The difference is that the GMRES residual is simply a `double`, rather than a vector.

4.5.1 Preconditioning

None of the linear system solvers discussed in the previous section preconditions the matrix. However, as described in Section 2.2.1, any linear system solver can be promoted to a preconditioned solver by replacing its matrix, A, with $\mathbf{A}' = \mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_2^{-1}$, its right hand side vector, b, with $\mathbf{b}' = \mathbf{M}_1^{-1}\mathbf{b}$, and the solution vector, x, with $\mathbf{x}' = \mathbf{M}_2\mathbf{x}$.

The class PreMtx implements the first of these substitutions; it represents the matrix A'. Its interface is shown in Figure 37. The three template arguments denote the type of M_1, A, and M_2, respectively. PreMtx supports the minimum number of functions required by the linear system solvers: vector multiplication and the member functions num_rows and num_cols.

The class PreLinSysSolver can be used to generate a preconditioned linear system solver from one of the unpreconditioned solvers. This is best illustrated using an example.

```
// Allocate the linear system
unsigned n = 100;
SpSkyMtx<double,double> m(n,n);
Vector<double> x(n), b(n);

< Assign values to m and b>

// Make a preconditioned GMRES solver
// Use tyepdef's to keep the template arguments manageable
typedef SpMtx<double,double> M;
typedef Vector<double> V;
typedef PreSpMtx<double,double> P;

GMRESSolver<PreMtx<P,M,P,V>,V> solver;
PreLinSysSolver<P,M,P,V> pre_solver(solver);

// Allocate and calculate a preconditioner
PreILU0SpSkyMtx<double,double> p(m);
p.make();

// Solve with left preconditioning
pre_solver.solve(&p,m,0,x,b);

// Solve with right preconditioning
pre_solver.solve(0,m,&p,x,b);
```

The constructor for the PreLinSysSolver class takes any LinSysSolver as an argument, provided that the matrix-type for this solver (i.e. its first template argument) is a PreMtx. In the example above a GMRESSolver was used. Consequently, pre_solver will solve the linear system using the GMRES method with ILU0 preconditioning.

The four template arguments for the PreLinSysSolver class are the types of, in order, the matrix M_1, the matrix A, the matrix M_2, and the vectors x and b. The types of the preconditioning matrices must agree with the types used in the template arguments of the LinSysSolver.

The full prototype of the solve member function is:

```
void PreLinSysSolver<LP,M,RP,V>::solve(const LP *p1, const M &m,
                                       const RP *p2, V &x, const V &b);
```

If either of the pointers p1 or p2 are null, then the corresponding preconditioning matrix is assumed to be the identity. Therefore, one has left preconditioning by setting p2 = 0 and right preconditioning by setting p1 = 0.

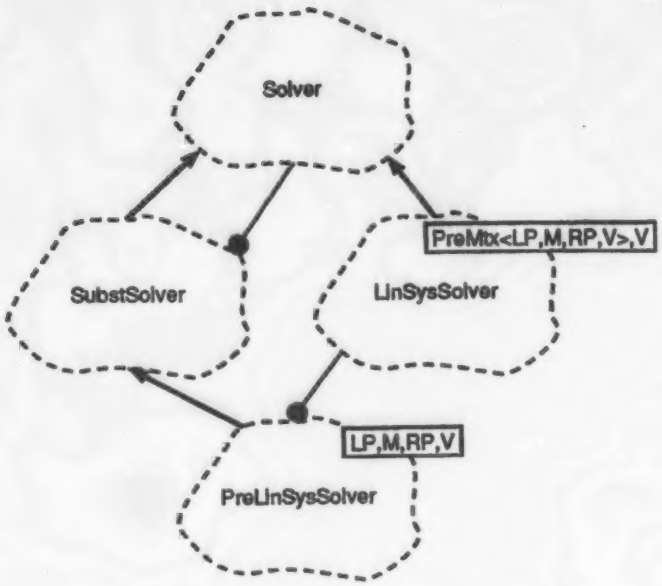

Figure 38: *Class diagram for* `PreLinSysSolver`.

The sole requirement of the preconditioning classes `LP` and `RP` is that a function with the following prototype be defined:

```
void solve(const LP &m, V &x, V &b);
```

This function solves the linear system $\mathbf{M} \cdot \mathbf{x} = \mathbf{b}$ for $\mathbf{x}$. If the linear system solver uses multiplication by the transpose of the matrix (e.g. the Quasi-Minimum Residual method), then the function

```
void solve(V &x, const LP &m, V &b);
```

must also be defined. It solves the linear system $\mathbf{x} \cdot \mathbf{M} = \mathbf{b}$ for $\mathbf{x}$. The class `PreSpMtx<ME,VE>` and its specializations fit this requirement when `V` is a `Vector<VE>`. It is possible that both versions of `solve` will alter the contents of `b`, so `b` must be copied before `solve` is called if it is to remain unchanged.

A class diagram for `PreLinSysSolver` is shown in Figure 38 and its interface in Figure 39. It is implemented as a `SubstSolver`, a specialization of the TRANSOM base class `Solver` that uses a pointer to a subsidiary solver which performs most of the work; therefore `SubstSolver` both inherits from `Solver` and has a pointer to a `solver` as a data member. In `PreLinSysSolver` the subsidiary solver is a `LinSysSolver` passed to the `PreLinSysSolver` via its constructor argument. The `solve` function adds the preconditioning steps (calculation of $\mathbf{b}'$ from $\mathbf{b}$, and calculation of $\mathbf{x}$ from $\mathbf{x}'$) to the solution provided by the `LinSysSolver`.

The class `PreLinSysSolver` is quite general, allowing matrix, vector, and preconditioner representations of many types provided that they adhere to the minimal requirements on the template arguments as described above. However, as can be seen by the example above, they

```
template<class LP, class M, class RP, class V>
class PreLinSysSolver: public SubstSolver
{
protected:
  typedef class LinSysSolver<PreMtx<LP,M,RP,V>,V> SolverType;
public:
  PreLinSysSolver(SolverType &s);
  virtual ~PreLinSysSolver() { }
  virtual void solve(const LP *p1, const M &m, const RP *p2, V &x,
                     const V &b);
};
```

Figure 39: *Interface for the class* `PreLinSysSolver`.

can become quite unwieldy to use due to the large number of template arguments. TRANSOM provides specializations of the linear system solvers for the case when the matrix is a specialization of `SpMtx`, the preconditioners are specializations of `PreSpMtx`'s, and the right hand side and solution vectors are `Vector`'s. For example, the class `SpSuccSubstSolver` is equivalent to

```
SpSuccSubstSolver<PreMtx< PreSpMtx<ME,VE>, SpMtx<ME,VE>, PreMtx<ME,VE>,
                          Vector<VE> >, Vector<VE> >
```

The definitions of `SpConjGradSolver`, `SpConjGradSqSolver`, `SpQMRSolver`, and `SpGMRESSolver` are similar.

The class `SpPreLinSysSolver` is equivalent to

```
PreLinSysSolver< PreSpMtx<ME,VE>, SpMtx<ME,VE>, PreSpMtx<ME,VE>,
                 Vector<VE> >
```

The `SpPreLinSysSolver` constructor argument can be any of the solvers `SpSuccSubstSolver`, `SpConjGradSolver`, etc. An example of the use of `SpPreLinSysSolver` was given in Section 3.4.

4.5.2 Linear System Solvers with Choice of Method

TRANSOM provides two linear system solvers which allow the choice of solution method to be specified by reading from an OFFSRF input file. The only difference in these classes is that one, `TransLinSysSolver`, assumes that multiplication by the matrix transpose is available while the other, `NoTransLinSysSolver`, does not. Therefore, `TransLinSysSolver` can offer the Quasi-Minimum Residual method as an option while `NoTransLinSysSolver` cannot.

The interfaces for `NoTransLinSysSolver` and `TransLinSysSolver` are shown in Figure 40. The data member `base_slv` is the unpreconditioned base solver; `pre_slv` is the preconditioned solver created using `base_slv`.

`NoTransLinSysSolver` maintains a single preconditioning matrix, `pmtx`, which it applies either on the left or the right according to the flag `preleft`. The flag `precond_made` keeps track

of whether the preconditioner has been calculated; if it has not been calculated when `solve` is called an error will result.

`NoTransLinSysSolver` is a specialization of the class `ChoiceSolver`. This is a generic TRANSOM class which allows a choice of solvers depending on input from an OFFSRF file. The `OFFSRF` class virtual function `read_record` is overloaded to read records from the input file and the virtual function `allocate_solver` is overloaded to allocate a solver according to the records found. In `NoTransLinSysSolver`, `read_record` is overloaded to understand records of the form:

```
{PRECONDITIONER: preconditioner-type
  {METHOD: preconditioning-method }
}
{SOLUTION METHOD: solution-method }
```

These records were described in Section 3.4.1.

4.5.3 The Classes `SpLinSys`, `NodeMapSpLinSys`, and `S2dSpLinSys`

Implementation of the class `SpLinSys` is straightforward. It maintains a right hand side vector, a solution vector, and allocates an appropriate specialization of `SpMtx` after the profile storage has been specified from the input file. A `TransLinSysSolver` is used to solve the linear system. `SpLinSys` is derived from the class `OFFSRF` so that it can read records from an OFFSRF input file. `SpLinSys` reads the MATRIX TYPE record itself (see Section 3.4.1), but it uses the `TransLinSysSolver` to read the PRECONDITIONER and SOLUTION METHOD records. The full interface is shown in Figure 41.

Implementation of the classes `NodeMapSpLinSys` and `S2dSpLinSys` is also straightforward. `NodeMapSpLinSys` is derived from `SpLinSys` by providing node mapped wrappers for the matrix and vectors, and OFFSRF functions to choose the type of element reordering. `S2dSpLinSys` is derived from `NodeMapSpLinSys`; it simply overloads the input functions to allow a wider range of choices for the matrix, preconditioner, and element order.

```
template<class M, class V, class P>
class NoTransLinSysSolver: public ChoiceSolver
{
protected:
  M *lhs;
  V *rhs, *soln;
  P *pmtx;

  LinSysSolver<PreMtx<P,M,P,V>,V> *base_slv;
  PreLinSysSolver<P,M,P,V> *pre_slv;

  int preleft;
  int precond_made;

public:
  NoTransLinSysSolver(M *mtx, V *rhs, V *s, P *p);
  virtual ~NoTransLinSysSolver();

  virtual void solve();
  void calc_preconditioner();
  int read_record(OFFSRF_ifstream &in, const Str &rec, int skip = 1);

protected:
  virtual void initialize();
  virtual void allocate_solver(const Str &name);
};

template<class M, class V, class P>
class TransLinSysSolver: public NoTransLinSysSolver<M,V,P>
{
public:
  TransLinSysSolver(M *mtx, V *rhs, V *s, P *p);
  virtual ~TransLinSysSolver() { }

protected:
  virtual void allocate_solver(const Str &name);
};
```

Figure 40: *Interfaces for the classes* NoTransLinSysSolver *and* TransLinSysSolver.

```
template<class ME,class VE>
class SpLinSys: public OFFSRF
{
protected:
  typedef TransMtxSolver< SpMtx<ME,VE>, Vector<VE>, PreSpMtx<ME,VE> >
          SpMtxSolver;

  int mtx_type;
  int precond_type;

  SpMtx<ME,VE> *lhs;
  Vector<VE> rhs, soln;
  PreSpMtx<ME,VE> *pmtx;
  SpMtxSolver *mslv;

public:
  SpLinSys(unsigned n);
  virtual ~SpLinSys();
  void solve();
  SpMtx<ME,VE>& get_mtx() { return *lhs; }
  Vector<VE>& get_rhs()   { return rhs; }
  Vector<VE>& get_soln()  { return soln; }
  int is_mem() const      { return lhs->is_mem(); }
  void get_mem();
  void free_mem();
  void calc_preconditioner();
  virtual int read_record(OFFSRF_ifstream&, const Str&, int skip = 1);
protected:
  virtual int allocate_mtx(const Str &type);
  virtual int allocate_pmtx(const Str &type);
  virtual int allocate_solver();
};
```

Figure 41: *Full interface for the class* SpLinSys.

5 Concluding Remarks

The C++ classes described in this memorandum provide a general framework for the use of sparse matrices and linear system solvers. While for some applications a wider range of profile storage schemes might be useful, those provided here have been found to work well in the DREA flow code TRANSOM.

Use of recent features of the C++ language, not available when these classes were developed, would result in cleaner and more general code. In particular, they would benefit from the exception handling mechanism and from template member functions.

Implementation of these classes in TRANSOM has improved it in several ways. The flow solvers all use an `SpMtx` with a storage scheme and solution method specified by the user in the TRANSOM input file. This results in a very loose coupling between the flow solvers and the linear systems used by them. Moreover, the single base class `SpMtx` is suitable for all the sparse matrices used in the code: it is no longer necessary to have separate matrix representations for the SPC and FE solvers. Indeed, although not yet implemented, it is now possible to use a single global matrix to solve the flow on neighbouring blocks which differ in structure (e.g. a structured block connected to an unstructured block). In addition, the SPC solvers are no longer constrained to have at most five elements on each matrix row. This allows for more general implementations of the boundary conditions and a straightforward extension of the linear system to three-dimensions, for which seven elements per row is usual.

References

1. D. Hally, "TRANSOM: A Multi-method Navier-Stokes Solver: Overall Design," DREA Technical Memorandum 97/231, 1997.

2. D. Hally, "Implementation of a Pseudo-Compressibility Navier-Stokes Solver in TRANSOM," DREA Technical Memorandum 98/239, 1998.

3. D. Hally, "Implementation of a Finite-Element Navier-Stokes Solver in TRANSOM," DREA Technical Memorandum 97/225, 1997.

4. C. Royal, "TRANSOM: Un solveur multi-bloc et multi-méthode pour les équations de Navier-Stokes," Bassin d'Essais des Carènes, Technical Report Etude 2560, No. 11, 1997.

5. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donatao, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Available from `templates@cs.utk.edu`.

6. D. J. Hall, D. W. Zingg, and C. R. Ethier, "A Laminar Incompressible Navier-Stokes Flow Solver," DREA Contractor Report CR/93/462, 1993.

7. D. Hally, "C++ Classes for Reading and Writing files in OFFSRF Format," DREA Technical Communication 94/302, 1994.

8. E. H. Cuthill, "Several Strategies for Reducing the Bandwidth of Matrices," in *Sparse Matrices and their Applications*, edited by D. J. Rose and R. A. Willoughby (Plenum Press, New York, 1972), pp. 157–166.

9. B. Stroustrup, *The C++ Programming Language*, Second Edition (Addison-Wesley Publishing Co., 1991).

10. G. Booch, *Object-Oriented Analysis and Design*, Second Edition (Addison-Wesley, 1994).

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document Organizations for whom the document was prepared, e.g. Establishment sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research Establishment Atlantic
P.O. Box 1012
Dartmouth, N.S. B2Y 3Z7

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

Unclassified

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

C++ Classes for Sparse Matrices and Iterative Linear System Solvers

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Hally, David

5. DATE OF PUBLICATION (month and year of publication of document)

October 1998

6a. NO OF PAGES (total containing information Include Annexes, Appendices, etc).

82

6b. NO. OF REFS (total cited in document)

10

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

DREA Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include the address).

Defence Research Establishment Atlantic
P.O. Box 1012
Dartmouth, N.S. B2Y 3Z7

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant).

1.g.a

9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written).

10a ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique to this document).

DREA TM/98/240

10b OTHER DOCUMENT NOS. (Any other numbers which may be assigned this document either by the originator or by the sponsor).

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(x) Unlimited distribution
( ) Distribution limited to defence departments and defence contractors; further distribution only as approved
( ) Distribution limited to defence departments and Canadian defence contractors; further distribution only as approved
( ) Distribution limited to government departments and agencies; further distribution only as approved
( ) Distribution limited to defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected).

Unlimited

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

C++ classes for representing sparse matrices are described. The matrices use one of four profile storage schemes: compressed row storage, skyline storage, sparse skyline storage (similar to skyline storage but with compressed arrays above and below the diagonal), and SdBlock storage (a special purpose storage scheme for matrices arising from partial differential equations on two-dimensional structured blocks).

C++ classes are also provided for iterative solution methods for linear systems of equations. The Conjugate Gradient, Conjugate Gradient Squared, Quasi-Minimum Residual, and Generalized Minimum Residual (GMRES) methods have been implemented. Element re-ordering and preconditioning using the diagonal (Jacobi) or incomplete LU factorization of order zero (ILU0) methods are also provided.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

C++
Sparse matrices
Linear systems